GIOVANI PIERI

SERVIÇO DE CONSENSO GENÉRICO TOLERANTE A INTRUSÕES PARA RESOLVER PROBLEMAS DE ACORDO

FLORIANÓPOLIS
2010

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

SERVIÇO DE CONSENSO GENÉRICO TOLERANTE A INTRUSÕES PARA RESOLVER PROBLEMAS DE ACORDO

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia
de Automação e Sistemas.

GIOVANI PIERI

Florianópolis, março de 2010.

SERVIÇO DE CONSENSO GENÉRICO TOLERANTE A INTRUSÕES PARA RESOLVER PROBLEMAS DE ACORDO

Giovani Pieri

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia de Automação e Sistemas, Área de Concentração em *Controle, Automação e Sistemas*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina.

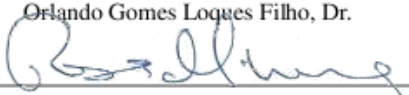
Joni da Silva Fraga, Dr.
Orientador

Lau Cheuk Lung, Ph.D.
Co-orientador

Eugênio de Bona Castelan Neto, Dr.
Coordenador do Programa de Pós-Graduação
em Engenharia de Automação e Sistemas

Banca Examinadora:

Orlando Gomes Loques Filho, Dr.



Rômulo Silva de Oliveira, Dr.

Leandro Buss Becker, Dr.

Aos meus pais.

AGRADECIMENTOS

Em primeiro lugar, gostaria de agredecer à minha família, principalmente meus pais, Ricardo e Cláudia, pelo incentivo que me deram desde a infância e sacrifícios que sei que fizeram para que hoje eu esteja aqui.
Agradeço ao meu professor orientador Joni da Silva Fraga e professor co-orientador Lau Cheuk Lung, sem os quais este trabalho não poderia ter sido realizado.
Gostaria também de agradecer aos amigos do Edugraf, em especial o professor Melgarejo e os colegas Diego, Ricardo, Pablo e Erich, que vem me acompanhando na caminhada acadêmica desde a graduação.
Por último, agradeço a todos os colegas e professores do DAS.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

SERVIÇO DE CONSENSO GENÉRICO TOLERANTE A INTRUSÕES PARA RESOLVER PROBLEMAS DE ACORDO

Giovani Pieri

Março/2010

Orientador: Joni da Silva Fraga, Dr.
Co-orientador: Lau Cheuk Lung, Ph.D.
Área de Concentração: Sistemas Computacionais.
Palavras-chave: Consenso, Algorítmos distribuídos, Tolerância a faltas, Tolerância a intrusão
Número de Páginas: xvii + 97

Esta dissertação descreve uma extensão ao Serviço de Consenso proposto por Guerraoui e Schiper. O objetivo é prover uma forma padronizada para implementar protocolos de acordo tolerantes a faltas bizantinas usando um serviço tolerante a faltas de intrusão construídos sobre tecnologias de virtualização. Para isto, implementamos um Serviço Genérico de Consenso (SGC). SGC separa as especificidades de diferentes problemas de acordo do consenso de uma forma clara, utilizando uma interação cliente-servidor, permitindo total independência entre protocolos de consenso utilizados e especializações específicas ao problema. Será mostrado o funcionamento do SGC, suas propriedades e como utilizá-lo para para resolver alguns problemas de acordo.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Automation and Systems Engineering.

SERVIÇO DE CONSENSO GENÉRICO TOLERANTE A INTRUSÕES PARA RESOLVER PROBLEMAS DE ACORDO

Giovani Pieri

Março/2010

Advisor: Joni da Silva Fraga, Dr.
Co-advisor: Lau Cheuk Lung, Ph.D.
Area of Concentration: Computational Systems
Keywords: Consensus, Distributed algorithm, Fault tolerance, Intrusion tolerance
Number of Pages: xvii + 97

This dissertation describes an extension of the Consensus Service proposed by Guerraoui and Schiper. The objective is to provide a standard way to implement agreement protocols resilient to Byzantine faults using an intrusion tolerant service built upon virtual machines technology. This is achieved through the implementation of a Generic Consensus Service (GCS). GCS separates specificities of different agreement problems from consensus in a clear way, using client-server interaction, allowing total independence between consensus protocols used and problem specific specializations. Besides that, the framework provides a set of properties and guarantees. It will be shown how the GCS works, its properties and how it may be used to solve some agreement problems.

Sumário

1 Introdução **1**
1.1 Motivação 1
1.2 Objetivos do Trabalho 2
1.3 Estrutura 3

2 Conceitos Básicos em Sistemas Distribuídos **5**
2.1 Modelo de Sistema 5
2.1.1 Processos 5
2.1.2 Canais de Comunicação 6
2.1.3 Modelo de Sincronismo 6
2.1.4 Modelo de Falhas 7
2.2 Problemas de Acordo 7
2.2.1 Consenso 8
2.2.2 Variações do Consenso Bizantino 9
2.2.3 Consenso de Vetor 10
2.2.4 Reliable broadcast 10
2.2.5 Atomic Broadcast 11
2.2.6 Group Membership 12
2.2.7 Non blocking atomic commitment (NBAC) 13
2.2.8 Replicação de Máquinas de Estados 14
2.3 Tecnologia de Virtualização 14
2.4 Considerações Finais 15

3 Trabalhos Relacionados **17**
3.1 Algoritmos de Consenso 17
3.1.1 Chandra e Toueg 17
3.1.2 Algoritmo com Detector de *Muteness* 19
3.1.3 Paxos 20
3.2 Serviços Replicados Tolerantes a Faltas Bizantinas 21
3.2.1 BFT 22
3.2.2 Zyzzyva 26
3.3 Modelos de Faltas Híbridos 28

3.3.1 TTCB . . . 29
3.3.2 A2M . . . 32
3.3.3 TrInc . . . 33
3.3.4 VM-FIT . . . 35
3.4 LBFT . . . 36
3.5 Servidor de Consenso no Modelo de *Crash* . . . 38
3.6 Considerações Finais . . . 41

4 Servidor de consenso **43**
4.1 Base Algorítmica do SGC . . . 45
4.2 Propriedades do SGC . . . 47
4.2.1 Lemas gerais . . . 47
4.2.2 Propriedades de Terminação . . . 49
4.2.3 Propriedades de Acordo . . . 51
4.2.4 Propriedades do Filtro de Consenso . . . 51
4.3 Aplicações . . . 52
4.3.1 Consenso de Vetor . . . 53
4.3.2 Strong Consensus . . . 54
4.3.3 Reliable Broadcast . . . 55
4.3.4 Atomic Broadcast . . . 58
4.3.5 Group Membership . . . 61
4.3.6 Non Blocking Atomic Commit . . . 65
4.4 Considerações Finais . . . 68

5 Protocolos de Acordo Baseados em Virtualização **71**
5.1 Componentes Confiáveis . . . 71
5.1.1 Postbox . . . 71
5.1.2 Postbox distribuída . . . 72
5.1.3 Algoritmos de Consenso . . . 73
5.1.4 Baseado em Postbox . . . 73
5.1.5 Baseado em Postbox Distribuída . . . 73
5.1.6 Corretude do algoritmo . . . 78

6 Experimentos e Detalhes de Implementação **83**
6.1 Detalhes de Implementação de Postbox . . . 83
6.1.1 Postbox . . . 83
6.1.2 Postbox Distribuída . . . 85
6.2 Protótipo . . . 87

7 Conclusão **89**

Lista de Figuras

2.1 Impasse quando há menos de $n = 3f+1$ processos. Processos cinza são bizantino. 9
2.2 Arquitetura de uma VMM tipo 1. 15
2.3 Arquitetura de um VMM tipo 2. 16

3.1 Troca de mensagens durante a execução normal do Paxos. . . 20
3.2 Troca de mensagens durante a execução normal do BFT. . . . 23
3.3 Troca de mensagens durante a execução normal do PBFT. . . . 27
3.4 Arquitetura de um sistema utilizando TTCB. 29
3.5 Arquitetura do sistema VM-FIT. 36
3.6 Troca de mensagens no LBFT. Linhas grossas indicam comunicação dentro da mesma máquina física. 37
3.7 Padrão de comunicação no serviço de consenso. São representadas mensagens enviadas por um cliente e um servidor apenas por motivos de clareza. 40

4.1 Arquitetura geral do Serviço Genérico de Consenso (SGC). . . 44
4.2 Padrão de comunicação de processos executando uma instância de consenso. 45

5.1 Execução boa do algoritmo de consenso baseado em *postbox* distribuída. 75

6.1 Implementação *postbox* através de memória compartilhada. . . 84
6.2 Implementação *postbox* através de arquivos compartilhados. . 85
6.3 Resultado de testes com quatro clientes, variando servidores . 87
6.4 Resultado de testes com três servidores, variando clientes . . . 87

Lista de Algoritmos

1 Algoritmo de consenso de Chandra e Toueg para o processo p 18
2 Algoritmo de consenso basedo no TTCB 31
3 Algoritmo de *multisend* 39
4 Iniciador no serviço de consenso de Guerraoui e Schiper. . . . 40
5 Cliente no serviço de consenso de Guerraoui e Schiper. 40
6 Servidor no serviço de consenso de Guerraoui e Schiper. . . . 41
7 Processo iniciador p_i do SGC 46
8 Processo cliente c_i do SGC 47
9 Processo servidor s_i do SGC 48
10 Função $Result$ do Problema Vetor de Consenso 53
11 Função $Result$ do Problema Strong Consensus 54
12 Cliente/Iniciador c_i do *Reliable Broadcast* 55
13 Filtro de Consenso do *Reliable Broadcast* 56
14 Cliente/Iniciador c_i do *Atomic Broadcast* 59
15 Filtro de Consenso para Resolução do *Atomic Broadcast* . . . 60
16 Algoritmo do cliente cm_i do *membership* 62
17 Algoritmo do coordenador c_i do problema do *group membership* 63
18 Algoritmo função result para problema de *membership* 64
19 Algoritmo do participante p_i do Atomic Commit 66
20 Algoritmo do coordenador c_i do problema de *Atomic Commit* . 67
21 Algoritmo função result para problema de commit distribuído . 67
22 Algoritmo de consenso utilizando a *postbox* 74
23 Algoritmo de consenso baseado em Postbox Distribuída 76
24 Implementação da *postbox* distribuída utilizando TrInc 86

Capítulo 1

Introdução

1.1 Motivação

Problemas de acordo, como *atomic commitment*, *group membership* e *total order broadcast* desempenham papel central em muitos sistemas distribuídos. Estes problemas compartilham uma característica em comum: requerem que cada processo concorde no resultado da computação distribuída. Normalmente, cada um destes problemas é tratado separadamente e resolvido por algoritmos especializados. Entretanto, em [1] foi proposto que um dos problemas de acordo mais simples, o problema de consenso, fosse usado como um paradigma para a construção de protocolos distribuídos. Um tempo depois esta proposição foi adequada para ambientes maliciosos [2, 3].

Em paralelo, uma outra tendência presente na literatura é o uso de modelos de faltas híbridos na construção de serviços tolerantes a faltas bizantinas [4, 5, 6, 7, 8]. Estes modelos consideram parte do sistema confiável e, portanto, sujeitos a hipóteses mais brandas de faltas. Com componentes confiáveis é possível tolerar até f processos faltosos dentre $n = 2f + 1$ processos e diminuir o número de passos necessários em um protocolo de acordo. Um dos problemas destas abordagens é o fato de requererem componentes confiáveis e, portanto, dependerem de medidas ou premissas que garantam a inviolabilidade destes componentes. O Serviço Genérico de Consenso (SGC) aqui proposto envolve a adoção também de modelo híbrido, mas sem a necessidade de que todos os nós de sistema mantenham as características de modelo híbrido de faltas. Sendo que apenas uma fração dos nós da rede possuem componentes confiáveis.

A literatura vem mostrando que para tolerar ataques maliciosos ao software, soluções necessitam adotar técnicas de replicação de estado em conjunto com técnicas de diversidade [9]. Estas propostas baseiam-se na observação de que faltas bizantinas com intenções maliciosas (intrusão) ocorrem em tentativas de explorar vulnerabilidades de porções do sistema computacional que são implementadas em software, como sistemas operacionais, drivers, daemons, aplicações, etc. Algumas das propostas recentes de sistemas tolerantes

à intrusão fazem uso da virtualização, assumindo muitas vezes esta tolerância em uma única máquina física [7, 10]. Nestas propostas, o sistema é replicado e cada réplica é executada em um ambiente virtualizado diferente sobre a mesma máquina física. Uma vantagem óbvia desta abordagem é que os custos de replicação para implementar serviços tolerantes à intrusão são reduzidos, pois apenas uma única máquina física é utilizada. Entretanto, a máquina é um ponto único de falha: se esta sofrer um *crash*, todos os serviços se tornam indisponíveis.

O uso da virtualização permite também facilmente a implementação do conceito de diversidade de serviços e de sistemas operacionais. Ou seja, a diversidade de projeto é usada para implementar, por exemplo, cada processo de um protocolo de consenso em uma máquina virtual distinta (com um sistema operacional distinto para cada VM). É importante ressaltar em defesa destes trabalhos, que se fixam somente na tolerância a intrusões e constroem suas abordagens em uma máquina física com redundâncias implementadas em várias maquinas virtuais, é que apesar da ocorrência de faltas bizantinas (intrusões) ser menos freqüente que faltas de *crash*, os danos causados pela primeira são mais severos que os danos causados pela última classe citada.

1.2 Objetivos do Trabalho

O objetivo deste trabalho é estender o Serviço de Consenso tolerante a faltas de *crash* apresentado por Guerraoui e Schiper [11]. A proposta é levar a um passo adiante e permitir que se construa a partir de um serviço, também genérico, protocolos tolerantes a intrusões. Ou seja, o objetivo do Serviço Genérico de Consenso proposto é prover uma infra-estrutura tolerante a intrusões que permita a construção de soluções para outros problemas de acordos mantendo as mesmas propriedades de tolerância a intrusões.

O *Serviço Genérico de Consenso* (SGC) proposto é baseado no uso de tecnologias de virtualização no sentido de prover meios para a tolerância a intrusões. Através da tecnologia de virtualização espera-se aumentar a resiliência do sistema através da adoção de um modelo hibrído de faltas.

Assim sendo, o segundo objetivo do trabalho é investigar e propor componentes confiáveis que quando adotados permitam um aumento da resiliência dos algoritmos de consenso. Deste objetivo nasce a necessidade de desenvolver algoritmos de consensos baseados em tais componentes.

Com base em tais objetivos gerais, pode-se citar os seguintes objetivos específicos:

1. Levantamento bibliográfico acerca de tolerância a faltas bizantinas, componentes confiáveis, algoritmos de consenso e problemas de acordo em geral, no intuito de formar uma base teórica sólida sobre a qual o trabalho possa ser desenvolvido.

2. Extender os algoritmos do serviço de consenso para que tolerem faltas bizantinas, levando em consideração qualquer mudança arquitetural que se faça necessária.

3. Provar a corretude do Serviço Genérico de Consenso.

4. Propor soluções algorítmicas para problemas de acordo, aplicando o Serviço Genérico de Consenso.

5. Provar a corretude de tais soluções baseadas no Serviço Genérico de Consenso.

6. Propor componentes confiáveis, definindo suas operações e propriedades.

7. Construir algoritmos de consenso para serem utilizados sobre tais componentes confiáveis.

8. Avaliar aspectos relativos a implementação dos componentes confiáveis.

9. Criar um protótipo que permita realizar uma análise do comportamento do Serviço Genérico de Consenso quando ocorre aumento de clientes ou servidores.

1.3 Estrutura

A dissertação continua no Capítulo 2 com a apresentação de conceitos básicos em sistemas distribuídos necessários para o desenvolvimento do trabalho, como por exemplo a definição de diversos problemas de acordo.

No capítulo 3, são apresentados alguns trabalhos descritos na literatura que influenciaram no desenvolvimento do Serviço Genérico de Consenso. Dentre estes trabalhos, encontra-se a descrição do trabalho de Guerraoui e Schiper que serviu como base para este trabalho.

O capítulo 4 introduz o Serviço Genérico de Consenso. São apresentados os algoritmos, teoremas de corretude e suas respectivas provas. Além disso, é apresentada uma série de aplicações do Serviço Genérico de Consenso onde vários problemas de acordo são resolvidos através dele.

No capítulo 5, dois componentes confiáveis são definidos, assim como suas primitivas e propriedades. Sobre estes componentes, um algoritmo de consenso para cada é construído.

Por último, no capítulo 6, são apresentados alguns aspectos acerca da implementação dos componentes confiáveis, a apresentação de um protótipo e resultados práticos obtidos rodando este protótipo em um ambiente controlado.

Capítulo 2

Conceitos Básicos em Sistemas Distribuídos

Este capítulo descreve alguns conceitos básicos em sistemas distribuídos que serão utilizados no decorrer do texto. Em especial, o modelo de sistemas e os diversos modelos de faltas que utilizados no decorrer do texto, detalhando as premissas assumidas em relação às máquinas virtuais utilizadas pelos algoritmos de acordo (apresentados no capítulo 5). Este capítulo também apresenta as definições de problemas de acordos recorrentes na literatura que serão implementados no contexto do serviço genérico de consenso proposto.

2.1 Modelo de Sistema

Ao projetar um sistema computacional distribuído tolerante a faltas, assumimos premissas relativas ao ambiente no qual a computação distribuída se desenrola, o comportamento dos diversos componentes que compõe este sistema e a forma como cada um destes falha (modelos de faltas). Este conjunto de premissas e definições compõe o modelo de sistema que adotamos. No que segue, detalharemos o modelo de sistema assumido neste trabalho. No modelo de sistema, mais de um modelo de faltas é apresentado, durante o texto é explicitado o modelos de faltas ao qual se está fazendo referência.

2.1.1 Processos

Assume-se que o sistema é composto por um conjunto Π ($\Pi = \{p_1, p_2, \ldots, p_n\}$) de processos, possivelmente infinitos. Este conjunto é divido em subconjuntos dependendo dos diferentes papéis assumidos pelos processos no servidor de consenso. Processos que assumem o papel de iniciadores pertencem a um subconjunto $I \subset \Pi = \{i_1, i_2, \ldots\}$, processos clientes ao subconjunto $C \subset \Pi = \{c_1, c_2, \ldots, c_{n_c}\}$, e processos servidores ao subconjunto $S \subset \Pi = \{s_1, s_2, \ldots, s_{n_s}\}$. Assume-se que $|C| = n_c$ e $|S| = n_s$ e $(C \cup I) \cap S = \emptyset$.

2.1.2 Canais de Comunicação

Assume-se que processos se comunicam por mensagens enviadas através de canais de comunicação. Quaisquer dois processos estão conectados por um canal de comunicação. Assume-se que estes canais de comunicação são ponto-a-ponto confiáveis e autenticados. Logo, mensagens enviadas não são modificadas, perdidas ou duplicadas pela infraestrutura da rede. Além disto, como o canal é autenticado, o emissor de uma mensagem m é conhecido pelo receptor de m.

Estes canais são de fácil implementação através do uso do protocolo TCP [12] aliados a técnicas criptográficas, como HMACS (Hashed MACs) [13] ou criptografia simétrica [14] em conjunto com sistemas de troca de chaves (ex: Diffie-Hellman [15]).

2.1.3 Modelo de Sincronismo

Diversos modelos de sincronismos são definidos na literatura, variando desde sistemas nos quais nenhuma premissa temporal é assumida, até sistemas nos quais todos os tempos no sistema são conhecidos [16, 17]. Estas premissas temporais levam em consideração tanto tempos de transmissão de mensagens na rede quanto o tempo de processamento de mensagens.

Sabe-se que consenso não possui solução de forma determinista em sistemas totalmente assíncronos na presença de uma única falta de *crash*, resultado obtido em [18]. Esta impossibilidade vem do fato de não se conseguir diferenciar, em um dado instante, se um processo sofreu *crash* ou apenas é lento. Assim, para assegurar que todas as propriedades do consenso sejam respeitadas, alguma sincronia deve ser assumida.

Adota-se neste trabalho o modelo de sistema com sincronia terminal (*eventually synchronous system model*) [16]. Neste modelo, em todas as execuções do sistema, existe um limite δ e um instante GST (*Global Stabilization Time*), tal que todas as mensagens enviadas em um instante $t > GST$ serão recebidas até o instante $t + \delta$. Note que tanto o instante GST quanto o valor δ não são conhecidos pelos processos e não necessitam serem o mesmo em diferentes execuções. Ou seja, na prática o sistema se comporta de forma assíncrona na maior parte do tempo, entretanto experimenta períodos de sincronia no qual mensagens podem ser enviadas e recebidas pelos processos, permitindo um avanço na computação distribuída.

Assume-se que todas as computações locais requerem intervalos de tempos desprezíveis. Esta premissa baseia-se na observação de que, apesar de algumas computações levarem tempo considerável para serem concluídas (ex. operações criptográficas), o aspecto assíncrono da computação distribuída torna menos relevante os processamentos locais, uma vez que estes estão menos sujeitos a interferências externas.

2.1.4 Modelo de Falhas

Processos em Π são classificados como corretos ou faltosos quando executam o algoritmo distribuído A. Processos corretos se comportam conforme especificado pelo algoritmo em todos os momentos de sua execução. O processos faltosos não necessariamente seguem a especificação algoritmica de A. Os principais tipos de falhas encontrados na literatura a que um processo está sujeito são [19]:

- **Parada (ou *crash*):** Um processo faltoso termina sua execução prematuramente. Nenhuma mensagem será enviada nem recebida por este processo.
- **Omissão de envio:** Um processo faltoso omite o envio de mensagens de maneira aleatória ou eventual.
- **Omissão de recepção:** Um processo faltoso omite a recepção de mensagens enviadas a ele de maneira aleatória ou eventual.
- **Bizantina:** O processo faltoso se comporta de forma arbitrária. Este tipo de falta também é conhecida como arbitrária ou maliciosa. Todas as formas anteriores de falhas estão englobadas nas faltas bizantinas.
- **Bizantina com autenticação:** O processo faltoso, assim como nas faltas bizantinas, se comporta de forma arbitrária. Entretanto, mecanismos criptográficos de autenticação estão disponíveis e não são forjáveis por processos faltosos. Desta forma, há uma simplificação na detecção de comportamentos bizantinos durante a execução da computação distribuída.

Recentemente, modelos de faltas híbridos vem sendo propostos na literatura [4, 5, 6, 7, 8, 20]. Estes modelos visam aproveitar-se de componentes confiáveis, sujeitos a um modelo de falta diferenciado do restante do sistema, para implementar algoritmos tolerantes a faltas bizantinas. Com isto, busca-se diminuir o número de processos necessário para garantir o bom funcionamento do algoritmo, ou a diminuição de passos de comunicação, e consequentemente, a latência.

Adota-se, nos algoritmos apresentados no capítulo 5, um modelo de faltas híbridos. Assim, parte do sistema utiliza o modelo de faltas bizantino enquanto outra adota modelo de crash.

2.2 Problemas de Acordo

Problemas de acordo, como *group membership*, *atomic broadcast* e *atomic commitment* são de suma importância em diversos sistemas distribuídos

[11, 21]. Todos estes problemas compartilham a característica de que os processos que tomam parte da computação distribuída devem, ao fim de seu processamento, chegar a um mesmo resultado.

Nesta seção, alguns problemas de acordo são apresentados e suas propriedades definidas. Algoritmos que solucionam alguns destes problemas utilizando o Serviço Genérico de Consenso são apresentados na seção 4.3.

2.2.1 Consenso

Dado um sistema distribuído composto por um número fixo e finito de processos independentes, no problema de consenso todos os processos corretos devem acabar por decidir o mesmo valor, que deve ter sido proposto por um dos processos do sistema. Formalmente este problema é definido por duas primitivas:

- $Propose(G, v)$: propõe o valor v para o conjunto de processos G.
- $Decide(v)$: executado pelo algoritmo de consenso quando um valor v é decidido. Normalmente, é utilizado para notificar aos interessados o valor decidido v.

Tais primitivas devem satisfazer as seguintes propriedades de *liveness* e *safety* [22]:

- **Acordo:** se dois processos corretos p e p' decidem v e v', respectivamente, então $v = v'$.
- **Validade:** um processo correto decide v se, e somente se, v foi previamente proposto por algum processo.
- **Terminação:** todo processo correto acaba por decidir.
- **Integridade:** um processo correto decide no máximo uma vez.

As propriedades de acordo, integridade e validade, no modelo de faltas de *crash*, podem ser uniformes. Neste caso, as restrições se aplicam a todos os processos, e não apenas a processos corretos. As propriedades de acordo e validade uniformes são definidas como segue:

- **Acordo Uniforme:** se dois processos p e p' decidem v e v', respectivamente, então $v = v'$.
- **Validade Uniforme:** um processo decide v se, e somente se, v foi previamente proposto por algum processo.
- **Integridade:** um processo decide no máximo uma vez.

Sob o modelo de faltas bizantino, o problema de consenso uniforme não pode ser resolvido [19, 23]. No modelo de faltas bizantinos processos faltosos podem falhar de forma arbitrária, logo não é possível impor um comportamento a processos faltosos, fator necessário em propriedades uniformes.

No modelo de faltas bizantino, a propriedade de validade exposta acima não é adequada, pois na presença de um processo faltoso qualquer valor pode ser decidido [24]. Várias propostas alternativas para a propriedade de validade são propostas na literatura, por exemplo *Strong Consensus* [25] e *Vector Consensus* [24].

Em [26] foi provado que para tolerar f processos faltosos são necessários $n = 3f+1$ processos. Este resultado advém do fato de que processos maliciosos podem mentir, isto é, um processo faltoso pode enviar uma mesma mensagem com valores distintos para processos diferentes. Um processo correto então não consegue decidir qual processo é o faltoso caso o $n \leq 3f$. A figura 2.1 ilustra o impasse com três processos, sendo um faltoso, que serve como base para a prova em [26], do ponto de vista do receptor 1, ele não sabe se o processo malicioso é o emissor ou o outro receptor. Em [4], um modelo de sistema acrescido de um componente confiável é utilizado para evitar este comportamento. Consequentemente, conseguiu-se aumentar a resiliência do algoritmo, sendo necessário $n \geq 2f + 1$ processos para tolerar f processos faltosos.

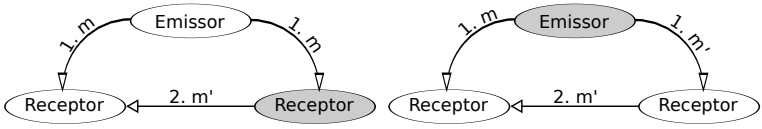


Figura 2.1: Impasse quando há menos de $n = 3f + 1$ processos. Processos cinza são bizantino.

2.2.2 Variações do Consenso Bizantino

O problema de consenso possui uma série de variações. Grande parte das variações são obtidas através de alterações na propriedade de validade do problema de consenso.

Além do problema de consenso exposto na subseção acima (2.2.1), conhecido como consenso multi-valor, existe o consenso binário. Neste consenso, processos podem propor e decidir apenas 0 ou 1. Todas as demais propriedades se mantém.

O *Strong Consensus* modifica a propriedade de validade para que, se todos os processos corretos propuserem o mesmo valor, este valor será esco-

lhido, caso contrário qualquer valor proposto poderá ser escolhido. As demais propriedades não são alteradas. Apesar de mais forte, esta propriedade não diz nada acerca do valor a ser escolhido no caso da existência de um processo faltoso.

A última variação do consenso que será abordada é introduzida em [24] e chamada de consenso com valor inicial certificado. Nesta variação, as propostas possuem um certificado sendo que a propriedade de validade garante que se um valor for decidido, ele foi proposto com um certificado válido. Esta variação é utilizada no Serviço Genérico de Consenso proposto.

2.2.3 Consenso de Vetor

O problema do consenso de vetor [24] é um problema de acordo no qual o valor de decisão é um vetor. Processos propõe valores e decidem um vetor que satisfaça as propriedades de validade de vetor. Sendo n o número de processos presentes, o vetor decidido possui n entradas, onde a i-ésima entrada do vetor corresponde a proposta realizada pelo processo i. Sendo f o número de processos faltosos, a propriedade de validade exige que as propostas de no mínimo $f + 1$ processos corretos estejam presentes no vetor decidido.

Este problema foi concebido para melhor adaptar o problema de consenso a um ambiente bizantino. O consenso e strong consenso não permitem que processos corretos detectem comportamentos maliciosos. A propriedade de validade do vetor de consenso possibilita, através da exigência de uma maioria de propostas corretas presentes no vetor, que processos corretos detectem comportamentos faltosos.

2.2.4 Reliable broadcast

O *Reliable Broadcast*[27] é um problema de comunicação de grupo no qual um processo p, pertencente a um conjunto de processos Π, deseja enviar uma mensagem m a todos os processos $p' \in \Pi$, de forma que todos os processos corretos entreguem a mesma mensagem m. No caso do processo p ser correto, todos os processos corretos deverão entregar m. Caso p seja faltoso, ou os processos corretos não irão entregar a mensagem ou uma mensagem m' qualquer será entregue por todos os processos.

O *Reliable Broadcast* é definido por meio de duas primitivas básicas[27]:

- $R - broadcast(G, m)$: utilizada para difundir a mensagem m entre todos os processos pertencentes ao grupo G.
- $R - deliver(p, m)$: chamada pelo protocolo de *Reliable Broadcast* para entregar à aplicação a mensagem m difundida pelo processo p.

Estas duas primitivas devem satisfazer as seguintes propriedades no problema do *Reliable Broadcast*[27]:

- **Validade:** Se um um processo correto p difundiu a m em um grupo G, então algum processo correto pertencente a G terminará por entregar m.

- **Acordo:** Se um processo correto pertencente a um grupo G entregar a mensagem m, então todos os processos corretos em G acabarão por entregar m.

- **Integridade:** Para qualquer mensagem m, todo processo correto irá entregar m no máximo uma vez. Caso o emissor de m seja correto, o emissor realizou a difusão de m no grupo G.

Em [27] faz-se uma simplificação, assume-se que toda mensagem possui um identificador atrelado, gerado pela concatenação do identificador do emissor p com um identificador único dentre as mensagens enviadas por p. Processos apenas iniciam o protocolo de *Reliable Broadcast* caso a origem da mensagem e os dados codificados em seu identificador sejam coerentes. Esta restrição será adotada nos protocolos aqui expostos e não acarreta perda de generalidade.

2.2.5 Atomic Broadcast

O problema de *Atomic Broadcast*[19], também conhecido como *Total Order Broadcast*, é um problema de comunicação de grupo. Assim como o *Reliable Broadcast*, através do *Atomic Broadcast* um processo p pertencente a um grupo G pode realizar a difusão de uma mensagem m entre todos os processos pertencentes ao grupo G de forma confiável. Isto é, processos corretos entregarão as mesmas mensagens, sendo que todas as enviadas por emissores corretos serão entregues. A diferença entre *Reliable Broadcast* e *Atomic Broadcast* se encontra no fato de que *Atomic Broadcast* faz asserções acerca da ordem em que as mensagens serão entregues à aplicação.

O *Atomic Broadcast* é definido por meio de duas primitivas básicas[19]:

- $Abroadcast(G, m)$: utilizada para difundir a mensagem m entre todos os processos pertencentes ao grupo G.

- $Adeliver(p, m)$: chamada pelo protocolo de *Atomic Broadcast* para entregar à aplicação a mensagem m difundida pelo processo p.

Além das três propriedades de validade, acordo e integridade do problema de *Reliable Broadcast*, o *Atomic Broadcast* é necessário garatir a propriedade de ordenamento [19]:

- **Ordem Total:** Se dois processos corretos p e p' entregam as mensagens m e m' difundidas no grupo G, então p entrega m antes de m' se, e somente se, p' entrega m antes de m'.

Foi mostrado em [22], para o modelo de faltas de *crash*, e em [2], para o modelo de faltas bizantinas, que o problema de *Atomic Broadcast* e consenso são equivalentes. Isto é, a partir de uma primitiva de consenso é possível criar algoritmos para solucionar o problema de *atomic broadcast*, e vice-versa. Isto vem a assegurar a possibilidade de utilizar o *Serviço Genérico de Consenso* como alicerce sobre o qual se venha construir um protocolo que resolva *Atomic Broadcast*.

2.2.6 Group Membership

No problema de *Group Membership* [28] processos em um sistema distribuídos acordam um conjunto de processos como operacionais. Neste sistema, um processo deve ser removido do grupo se for faltoso (ou suspeito). Um processo deve ser adicionado ao grupo, se for recuperado ou se foi erroneamente suspeito de ser faltoso. O protocolo de *Group Membership* garante que estas mudanças são percebidas pelos processos consistentemente.

Em [28] e no que segue, assume-se que todo processo é munido de um detector de falhas [22, 24, 29, 30, 31] que classifica processo do sistema em faltosos e corretos. As suspeitas dos detectores de falhas podem ser equivocadas e contraditórias, isto é, dois processos corretos podem discordar quando a condição de um processo.

O protocolo de *Group Membership* progride em visões. Uma visão V_j é o conjunto de processos assumidos como corretos pelo processo j. O protocolo atualiza V_j baseado nas informações trocadas entre os processos e deve garantir que dois processos corretos percebam as mesmas mudanças de visão. No que segue, as visões são referenciadas por V_j^x, sendo x o número de sequência das visões indicando a ordem que o processo as percebeu. Por exemplo, a primeira visão instalada pelo processo j é V_j^0, a segunda visão é V_j^1, a terceira V_j^2, e assim sucessivamente. Ao criar uma nova visão V_j^x, diz-se que a visão foi instalada. A primeira visão instalada, V_j^0, em qualquer cliente correto, é igual a um mesmo conjunto finito de processos.

Em um ambiente bizantino, um protocolo de membership deve satisfazer as propriedades [28]:

- **Singularidade (*Uniqueness*):** Se p_i e p_j são corretos e V_i^x e V_j^x foram instaladas, então $V_i^x = V_j^x$.

- **Validade:** Se p_i é correto e V_i^x foi instalada, então $p_i \in V_i^x$ e todo processo correto $p_j \in V_i^x$, acabará instalando V_j^x.

- **Integridade:** Se $p \in V^x - V^{x+1}$, então algum processo correto $q \in V^x$ classificou p como faltoso. Se $p \in V^{x+1} - V^x$, então algum processo correto $q \in V^x$ classificou p como correto.

- **Vivacidade:** Se existe um processo correto $p \notin V^x$ tal que $\lceil (2|V^x| + 1)/3 \rceil$ membros corretos de V^x não suspeitam de p, ou um $q \in V^x$ tal que $\lfloor (|V^x| - 1)/3 \rfloor + 1$ membros corretos de V^x suspeitam de q, então V^{x+1} acabará sendo instalada em algum processo correto.

2.2.7 Non blocking atomic commitment (NBAC)

Uma transação distribuída é uma transação que engloba vários nós em uma rede de computadores, mantendo as propriedades ACID (atomicidade, consistencia, integridade, durabilidade) [32]. Uma das questões mais interessantes neste cenário é garantir a atomicidade, isto é, ou todos os processos realizam com sucesso a operação ou nenhum o realiza, como se a operação nunca existisse. Tal problema é conhecido como *atomic commitment* [33, 34].

Aqui dividiremos os processos pertencentes ao sistema distribuído em dois papéis: coordenadores e participantes. Os coordenadores recebem votos dos participantes e decidem se estes devem executar a operação ou abortá-la. Os participantes são aqueles processos que desejam executar uma operação de forma transacional.

É impossível resolver o problema NBAC sob o modelo de faltas bizantinas. Este fato advém da impossibilidade de resolver o problema de consenso uniforme sob tal modelo de faltas [23].

Detectores de falhas *?P* foram definidos em [29], onde se provou que são necessários para a resolução do problema NBAC. Estes detectores indicam se uma falha no sistema ocorreu, obedecendo as seguintes propriedades:

- ***Anonymous Completeness***: caso um processo falhe, existe um tempo a partir do qual todo processo correto detecta uma falha permanentemente.

- ***Anonymous Accuracy***: nenhum falha é detectada até que um processo tenha falhado.

O problema de NBAC satisfaz as seguintes propriedades:

- **Acordo:** Se dois participantes distintos decidem v e v' respectivamente, então v=v'.

- **Validade-*abort*:** *Abort* é a única decisão possível se algum processo vota não.

- **Validade-*commit*:** *Commit* é a única decisão possível se todos os processos são corretos e votam sim.

- **Vivacidade:** Todo participante correto acabará decidindo algum valor.

2.2.8 Replicação de Máquinas de Estados

A Replicação de Máquinas de Estados [35] é uma técnica utilizada para tornar um serviço qualquer, representado através de uma máquina de estados determinística, tolerante a faltas de crash [35] ou bizantinas [36]. Nesta abordagem, o provedor do serviço é replicado e o protocolo de replicação de máquinas de estados é responsável por garantir a consistência entre as réplicas, obrigando-as a evoluir pelos mesmos estados.

De forma a garantir que as réplicas evoluam da mesma forma, réplicas partindo de um mesmo estado, executando o mesmo conjunto de requisições na mesma ordem, chegam ao mesmo estado final. Tal comportamento é conhecido como determinísmo de réplica, e para ser obtido o serviço deve ser modelado como uma máquina de estados determinística.

O problema de replicação de máquina de estados pode ser visto como um problema de acordo. Isto porque, dado um conjunto de réplicas partindo do mesmo estado inicial, as réplicas devem acordar quais requisições atender e em que ordem. Todas as réplicas então, executam as requisições acordadas.

2.3 Tecnologia de Virtualização

Virtualização é uma tecnologia que permite particionar recursos computacionais entre uma ou mais máquinas virtuais. Tal tecnologia surgiu no fim da década de 60 como uma camada de software sobre a máquina física capaz de rodar aplicações sem moficações dentro de uma máquina virtual. Assim, o aparato computacional extremamente caro era compartilhado de forma segura e simples entre diversos usuários [37, 38].

Uma das premissas básicas encontrada na tecnologia de virtualização é o isolamento das máquinas virtuais. Máquinas virtuais que operam sobre um mesmo gerenciador de máquina virtual devem ser independentes, de forma que a falha de uma máquina não cause nenhuma consequência em outras máquinas virtuais.

Sistemas virtualizados são normalmente classificados em dois tipos, dependendo da sua arquitetura [39]. Em sistemas virtualizados do *tipo 1* (figura 2.2) sobre o *hardware* opera um gerenciador de máquinas virtuais, também conhecido como *hypervisor*, responsável por criar e gerenciar as diversas máquinas virtuais. Por sua vez, cada máquina virtual roda sistema operacional e aplicações. Como exemplo de sistema que adota esta abordagem, pode-se citar o sistema de virtualização Xen [40].

Em sistemas virtualizados do *tipo 2*(figura 2.3) o gerenciador de máquinas virtuais opera sobre um sistema operacional de uso geral, ao lado de aplicações de usuário. Como exemplo de sistema que adota esta abordagem, pode-se citar os sistemas de virtualização VirtualBox [41] e VMware [42].

Assume-se que o sistema operacional hospedeiro, onde o monitor MMV

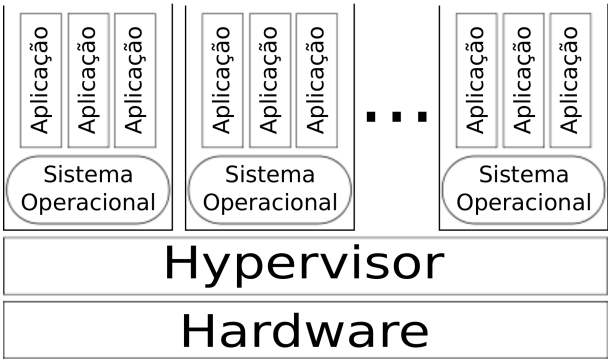


Figura 2.2: Arquitetura de uma VMM tipo 1.

executa, pode ter vulnerabilidades que poderiam ser usadas por intrusos (atacantes). Entretanto, estas não podem ser exploradas através da rede na nossa proposta. Esta hipótese é factível, pois assumimos como premissa que o sistema operacional hospedeiro não será acessível através da rede. Isto pode ser facilmente implementado em sistemas reais utilizando sistemas de *firewall* ou desabilitando/removendo *drivers* de rede do sistema operacional hospedeiro. Apenas VMs possuem acesso a interfaces de rede.

De forma a dificultar exploração de falhas encontradas em software, pode-se empregar a diversidade de serviços e de sistemas operacionais. Esta abordagem é facilitada pela utilização da tecnologia de virtualização, pois permite que, por exemplo, que cada processo de um protocolo de consenso se encontre em uma máquina virtual distinta utilizando diferentes sistemas operacionais. Esta prática diminui a probabilidade de que uma vulnerabilidade possa ser explorada em todas as máquinas virtuais dentro de uma janela de vulnerabilidade, período no qual não pode existir mais do que uma fração de máquinas faltosas simultaneamente.

2.4 Considerações Finais

Neste capítulo foram introduzidos os conceitos básicos em sistemas distribuídos sobre os quais o trabalho se fundamentou. Além disso, o modelo de sistema que foi especificado, com exceção do modelo de faltas que propositalmente foi deixado em aberto. Esta abordagem foi escolhida pois no decorrer das discussões modelos de faltas diferentes são adotados.

Na segunda parte do capítulo foram introduzidos alguns problemas de acordo, problemas estes que são passíveis de implementação através do *framework* proposto. Problemas de acordo são fundamentais em sistemas dis-

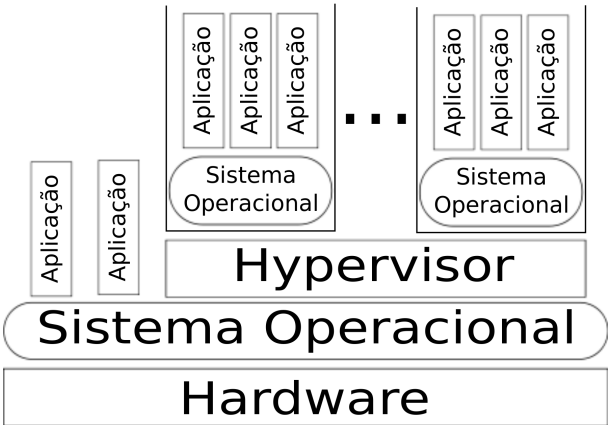


Figura 2.3: Arquitetura de um VMM tipo 2.

tribuídos e possuem as mais diversas aplicações, sendo utilizados para prover tolerância a faltas de sistemas distribuídos, primitivas de comunicação de grupo, aspectos transacionais (essenciais em sistemas de banco de dados distribuídos), entre outras aplicações.

Por fim, uma pequena introdução a tecnologia de virtualização foi realizada. Salientando-se as premissas utilizadas quando o adotamos nas soluções tolerantes a faltas posteriormente apresentadas.

Capítulo 3

Trabalhos Relacionados

Este capítulo descreve alguns trabalhos encontrados na literatura que formaram a base sobre o qual trabalho apresentado neste texto foi elaborado. Aborda-se alguns algoritmos de acordo clássicos encontrados na literatura, algoritmos de replicação de máquinas de estado tolerantes a faltas bizantinas, algoritmos e arquiteturas que utilizam modelo hibrídos de faltas. Por fim, introduz-se o serviço de consenso introduzido por Guerraoui e Schiper.

3.1 Algoritmos de Consenso

O problema de consenso apresentado na seção 2.2.1 foi muito estudado na literatura. Vários algoritmos foram propostos, dentre eles, três algoritmos de consenso sendo que dois deles utilizam o modelo de faltas de *crash*, enquanto um adota o modelo de faltas bizantinas.

3.1.1 Chandra e Toueg

Em [22], Chandra e Toueg propõem algoritmos para a resolução de consenso no modelo de faltas de *crash* em ambientes assíncronos. Devido a impossibilidade da resolução determinística de consenso em um ambiente assíncrono na presença de faltas de *crash* [18], Chandra e Toueg propõem o acréscimo de um oráculo, conhecidos por detectores de falhas, ao sistema distribuído capaz de prover informações acerca de processos faltosos e corretos.

Detectores de falhas classificam processos em corretos e suspeitos de falha, fornecendo ao algoritmo uma lista de processos faltosos. Detectores são definidos por duas propriedades: precisão e completude. São apresentadas duas variações para a propriedades de completude (forte e fraca) e quatro variações da propriedade de precisão (fraca, forte, *eventually weak* e *eventually strong*), dando origem a oito diferentes detectores de falhas.

O algoritmo 1, apresentado em [22], utiliza detector de falha $\Diamond S$ para resolver consenso. Tal detector possui propriedade de (i) completude forte e (ii) precisão *eventually weak*, isto é, (i) todo processo que sofre falta de *crash* será suspeito permanentemente por todo processo correto, e (ii) existe um instante a partir do qual um processo correto nunca será suspeito de ser

faltoso por nenhum processo correto.

Algorithm 1 Algoritmo de consenso de Chandra e Toueg para o processo p

1: **procedure** PROPOSE(v_p)
2: $estimate_p \leftarrow v_p$
3: $state_p \leftarrow undecided$
4: $faulty_p \leftarrow$ saída do detector de falhas
5: $r_p \leftarrow 0$
6: $ts_p \leftarrow 0$
7: **while** $state_p = undecided$ **do**
8: $r_p \leftarrow r_p + 1$
9: $c_p \leftarrow (r_p \ mod \ n) + 1$
10: **send** $\langle p; r_p; estimate_p; ts_p \rangle$ **to** c_p
11: **if** $p = c_p$ **then**
12: **wait until** $\lceil \frac{n+1}{2} \rceil$ processos $q : p$ recebeu $\langle q; r_p; estimate_q; ts_q \rangle$ de q
13: $msgs_p[r_p] \leftarrow \{\langle q; r_p; estimate_q; ts_q \rangle : p$ recebeu $\langle q; r_p; estimate_q; ts_q \rangle$ de $q\}$
14: $t \leftarrow$ maior ts_q tal que $\langle q; r_p; estimate_q; ts_q \rangle \in msgs_p[r_p]$
15: $estimate_p \leftarrow$ um $estimate_q : \langle q; r_p; estimate_q; t \rangle \in msgs_p[r_p]$
16: **send** $\langle p; r_p; estimate_p \rangle$ **to** all
17: **end if**
18: **wait until** recebeu $\langle c_p; r_p; estimate_{c_p} \rangle$ de c_p **or** $c_p \in faulty_p$
19: **if** recebeu $\langle c_p; r_p; estimate_{c_p} \rangle$ de c_p **then**
20: $estimate_p \leftarrow estimate_{c_p}$
21: $ts_p \leftarrow r_p$
22: **send** $\langle p; r_p; ack \rangle$ **to** c_p
23: **else**
24: **send** $\langle p; r_p; nack \rangle$ **to** c_p
25: **end if**
26: **if** $p = c_p$ **then**
27: **wait until** $\lceil \frac{n+1}{2} \rceil$ processos $q : p$ recebeu $\langle q; r_p; ack \rangle$ ou $\langle q; r_p; nack \rangle$ de q
28: **if** $|\{m$ recebida de $q : m = \langle q; r_p; ack \rangle\}| = \lceil \frac{n+1}{2} \rceil$ **then**
29: **send** $\langle p; r_p; estimate_p \rangle$ **to** all
30: **end if**
31: **end if**
32: **end while**
33: **end procedure**

Este algoritmo exige que a maioria de processos sejam corretos para garantir a propriedade de terminação do problema de consenso. Ele opera em rodadas assíncronas, sendo que cada uma possui como líder o processo com identidade $(r \ mod \ n) + 1$, sendo r o número da rodada e n o número de processos realizando o consenso. Uma rodada assíncrona inicia com o envio por parte dos processos de suas propostas ao líder da rodada. Então, o líder seleciona uma das propostas com maior estampilha de tempo ts e a envia aos demais processos. Ao receberem a mensagem do líder com a proposta, os processos trocam a sua proposta pela proposta recebida do líder da rodada, atualizam a estampilha de tempo ts para a atual rodada e enviam uma mensagem ack ao

líder. Caso um processo suspeite que o líder é faltoso, uma mensagem $nack$ é enviada ao líder. O líder coleta as mensagens de ack e $nack$, decidindo a proposta atual caso a maioria dos processos enviem ack.

O algoritmo funciona pois ao receber uma maioria de mensagens ack, uma maioria de processos adotou o valor v proposto pelo líder. A partir deste momento a proposta do líder foi chaveada, isto é, nenhum outro líder irá propor um valor diferente de v, devido a estampilha de tempo ts.

Algoritmos para a resolução de consenso utilizando outros detectores de falhas não serão apresentadas, mas podem ser encontrados em [22, 31, 43, 44, 45].

3.1.2 Algoritmo com Detector de *Muteness*

Detectores de *muteness* [46] é uma extensão de detectores de falhas de Chandra e Toueg para o modelo de faltas bizantino. Entretanto, a natureza de faltas bizantinas é diferente da natureza das faltas de *crash*, pois faltas bizantinas englobam faltas específicas da aplicação. Logo, a natureza totalmente modular que se conseguiu alcançar em detectores de falhas em *crash* não é possível em detectores de falhas bizantinos. Quando nós estão sujeitos apenas a falhas de *crash*, detectores de falhas podem ser implementados de forma modular já que estes devem determinar apenas se um dado nó de rede está ativo. Entretanto, um nó sujeito a falhas bizantinas pode-se comportar de forma arbitrária. Logo, o conhecimento de quais mensagens devem ser enviadas e recebidas por cada nó da rede deve ser conhecido pelo detector, fazendo com que o mesmo se torne intimamente relacionado com o algoritmo que o está utilizando.

Um processo é considerado mudo quando não envia uma mensagem previamente expressada no algoritmo em questão. Detectores de *muteness* não detectam todos os processos faltosos, mas apenas processos mudos. Outras falhas devem ser tratadas pelo próprio algoritmo. Além disso, o detector de falhas deve conhecer alguns detalhes do algoritmo de forma a reconhecer instantes nos quais mensagens são esperadas de outros processos e assim, detectar processos mudos. Detectores de muteness $\Diamond M$ possuem duas propriedades: completude e precisão. A propriedade de completude diz que todo processo correto detecta de forma permanente processos mudos para com ele. A propriedade de precisão garante que existe um instante a partir do qual um processo correto não é suspeito de ser mudo por nenhum outro processo correto.

O algoritmo de consenso apresentado em [46] não será explicitado aqui. Tal algoritmo baseia-se no paradigma de líder rotativo e avança em rodadas assíncronas. Cada rodada, por sua vez, é dividida em duas fases. Durante a primeira fase o algoritmo tenta decidir o valor proposto pelo líder. Para isto,

primeiramente o líder realiza o broadcast de uma mensagem contendo a sua proposta. Os processos então reenviam o broadcast para todos e caso uma maioria possua a mesma estimativa após estes passos, uma decisão pode ser tomada. A segunda fase entra em cena quando o líder é suspeito de ser bizantino por uma maioria de processos. Neste caso, ações são tomadas para assegurar que se um processo decidiu v em uma rodada r, todos os processos corretos iniciam a rodada $r + 1$ com a estimativa v.

3.1.3 Paxos

O algoritmo de paxos [47, 48] resolve o problema de consenso no modelo de faltas de omissão. É um algoritmo eficiente e serviu de base para o desenvolvimento do algoritmo de acordo tolerante a faltas bizantinas utilizado pelo PBFT [36].

No algoritmo de paxos, três papéis distintos são definidos: proponentes, aceitadores (*acceptors*) e aprendizes. Os proponentes propõem valores aos aceitadores, estes por sua vez escolhem uma única proposta em conjunto e fazem com que os aprendizes aprendam este valor. Estes papéis podem ser mapeados a processos de qualquer maneira, inclusive com todos os processos assumindo todos os papéis.

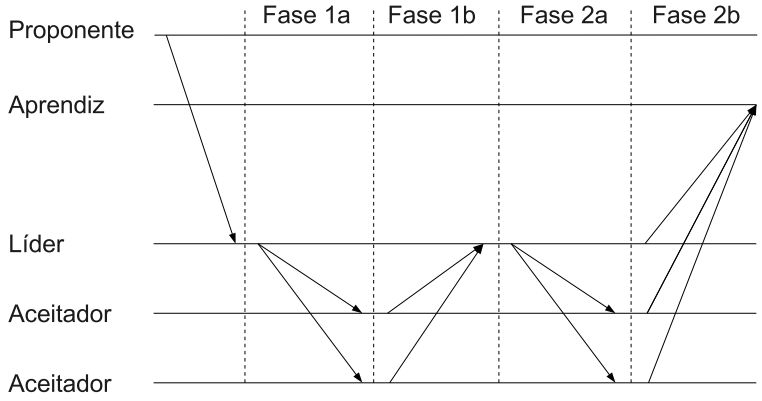


Figura 3.1: Troca de mensagens durante a execução normal do Paxos.

O algoritmo de paxos se desenrola em rodadas assíncronas, sendo que cada rodada assíncrona possui um identificador atrelado e um coordenador pré-determinado. Cada rodada é dividida em duas fases, como pode ser visto na figura 3.1. Durante a primeira fase o coordenador da rodada tenta fazer com que uma maioria de processos participem da rodada da qual é líder enviando uma requisição. Cada aceitador então responde ao líder informando da sua

participação. Caso uma maioria de aceitadores informem ao líder que participarão de rodada por ele liderada, o líder escolhe uma proposta e a envia a todos os aceitadores. A escolha desta proposta é feita baseando-se nas mensagens de confirmação de participação da rodada por ele recebidas, de forma que caso um processo tenha decidido v em alguma rodada anterior o líder proponha v.

O coordenador mantém duas informações: a rodada com maior identificador que ele coordenou ($crnd[c]$) e o valor ($cval[c]$) escolhido para a rodada $crnd[c]$ ou $\perp$ caso nenhum valor tenha sido escolhido. O aceitador a mantém as seguintes informações: a maior rodada que o aceitador participou ($rnd[a]$), a maior rodada na qual o aceitador votou ($vrnd[a]$) e o valor que o aceitador votou na rodada $vrnd[a]$ ($vval[a]$). A rodada i com coordenador c se desenrola da seguinte forma [49]:

1. (a) Se $crnd[c] < i$ então c inicia a rodada i fazendo $crnd[c] \leftarrow i$, $cval[c] \leftarrow \perp$ e enviando uma mensagem para cada aceitador requisitando a sua participação na rodada i.

   (b) Se um aceitador a recebe uma requisição para participar da rodada i e $i > rnd[a]$, então a faz $rnd[a] \leftarrow i$ e envia ao coordenador c uma mensagem contendo o número da rodada i e os valores $vrnd[a]$ e $vval[a]$. Caso $i \leq rnd[a]$, a ignora a requisição através de uma mensagem $nack$.

2. (a) Se $crnd[c] = i$, $cval[c] = \perp$ e c recebeu mensagens da fase 1b referentes a rodada i de uma maioria de aceitadores, então c utiliza estas mensagens para escolher um valor v, faz $cval[c] \leftarrow v$ e envia uma mensagem a todos os aceitadores requisitando que eles votem na rodada i para aceitar v.

   (b) Se um aceitador a recebe uma requisição para votar em uma rodada i para aceitar um valor v, $i \geq rnd[a]$ e $vrnd[a] = i$, então a vota na rodada i pelo aceite de v, fazendo $vrnd[a] \leftarrow i$ e $rnd[a] \leftarrow i$ e $vval[a] \leftarrow v$ e envia uma mensagem para todos os aprendizes anunciando o seu voto na rodada i. Caso $i < rnd[a]$ ou $vrnd[a] = i$, a ignora a requisição.

Várias variações e otimizações para o algoritmo original explanado acima são encontrados na literatura [49, 50, 51, 52, 53, 54]. Em [3, 36], versões tolerantes a faltas bizantinas são descritas.

3.2 Serviços Replicados Tolerantes a Faltas Bizantinas

Algoritmos de replicação de serviços visam tornar serviços quaisquer, modelados como máquinas de estados determinístas, em serviços tolerantes a faltas bizantinas. O trabalho seminal em PBFT [36] foi, em grande parte,

responsável por trazer interesse em tolerância a faltas bizantinas novamente a comunidade científica, após alguns trabalhos iniciais nos anos 80 e 90 [28, 43, 55, 56], como a proposição do problema dos generais bizantinos [26].

Após o algoritmo de replicação PBFT, diversos trabalhos visando tratamento de faltas bizantinas surgiram. Dentre os algoritmos de replicação que seguiram, encontram-se o Zyzzyva [57] e mais recentemente Spin [58]. No que segue, uma breve descrição de cada um destes algoritmos será apresentada.

3.2.1 BFT

O sistema BFT (*Byzantine Fault Tolerance*) [36] propõe um protocolo cujo objetivo é prover serviços tolerantes a faltas bizantinas através da técnica de replicação de máquinas de estado. Este sistema tem como objetivo que o sistema possa ser utilizado na prática e, portanto, possui uma série de otimizações. Dentre as otimizações realizadas, está a utilização de MACs ao invés de criptografia assimétrica para autenticação de mensagens e mecanismo de recuperação pró-ativa. Isto é, réplicas são recuperadas a um estado não faltoso periodicamente, mesmo que não haja suspeitas de que tal réplica seja faltosa. Com esta técnica, o protocolo é capaz de tolerar um número qualquer de faltas durante a sua execução, desde que o limiar de f réplicas faltosas não seja ultrapassado dentro de um intervalo de tempo chamado *janela de vulnerabilidade*.

O serviço é implementado por n réplicas, dentre as quais no máximo f são faltosas. Os algoritmos garantem corretude independentemente do número de clientes faltosos que utilizam o serviço replicado. A garantia dada em presença de clientes faltosos é de que todas as ações realizadas por um cliente faltoso são vistas de uma forma consistente por servidores corretos. Além disso, ações de clientes faltosos são limitadas por um sistema de autenticação e autorização, logo um cliente só poderá realizar uma operação sobre o serviço caso esteja autorizado a fazê-lo.

PBFT provê corretude e vivacidade assumindo que não existam mais do que f réplicas faltosas durante a execução do serviço. Corretude é provida em ambientes assíncronos, enquanto as propriedades de vivacidade exigem sincronia terminal. Assim, clientes acabam recebendo respostas as suas requisições caso: no máximo f réplicas sejam faltosas e $delay(t)$ não cresça mais rapidamente que t indefinidamente, onde: $delay(t)$ é o tempo entre o momento t quando a mensagem é enviada pela primeira vez e o momento em que a resposta é recebida. Isto garante a sincronia terminal.

A grosso modo, o algoritmo opera da seguinte forma: clientes do serviço replicado enviam suas requisições para a execução de uma determinada operação a todas as réplicas. As réplicas não-faltosas decidem uma única or-

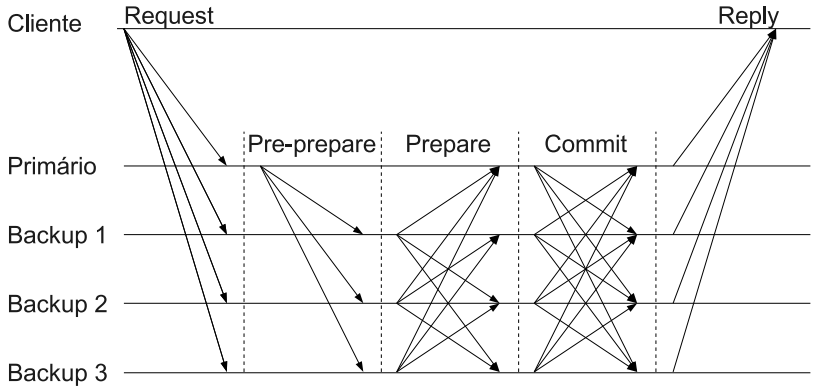


Figura 3.2: Troca de mensagens durante a execução normal do BFT.

dem de execução para as operações pendentes. Como a máquina de estados implementadas nas réplicas corretas é determinística e iniciam no mesmo estado, todas as réplicas corretas evoluem da mesma forma obtendo os mesmos resultados para todas as operação requisitadas. O cliente espera a chegada de $f + 1$ respostas idênticas, pois quando isto acontecer o cliente é capaz de inferir que existe uma réplica correta que lhe enviou uma resposta, e portanto tal resposta está correta.

Para decidir a ordem de execução das requisições, é utilizado um mecanismo primário-backup. As réplicas se organizam em uma sucessão de configurações chamadas visões, que são numeradas seqüencialmente. Em cada visão uma réplica, eventualmente faltosa, assume o papel de primário, enquanto as outras assumem o papel de backup. O primário é responsável por ordenar as requisições feitas pelos clientes e propô-la às réplicas backup. O primário de uma determinada visão v é a réplica de número $p = v \ mod \ n$. Esta escolha do primário assegura que uma réplica não-faltosa assuma o papel de primário em no máximo f trocas de visão, evitando que réplicas faltosas monopolizem o papel de primário.

Durante uma operação sem faltas o protocolo executa em três fases: *pre-pepare*, *prepare* e *commit*. O objetivo das fases de *pre-pepare* e *prepare* é realizar a ordenação total das operações a serem executadas dentro de uma dada visão v, enquanto as fases de *prepare* e *commit* asseguram ordenação total entre visões distintas. Na fase *pre-prepare* o primário atribui um número de sequência a uma requisição do cliente e o envia a todos os processos. Uma réplica i aceita a proposta caso o número ainda não tenha sido utilizado em

uma mensagem *pre-prepare* v aceita por i na visão v e tal número esteja dentro de um intervalo $[h, H]$, onde h e H são conhecidos como marca d'água baixa e alta, respectivamente. O sistema de marca d'água tem o objetivo de evitar que um primário malicioso esgote os números de sequência passíveis de serem atribuídos a mensagens e permite ao coletor de lixo, isto é, o descarte de mensagens e dados não mais necessários. O aceite da mensagem *pre-prepare*, significa que a réplica está de acordo com o número de sequência atribuído pelo primário e tal fato é comunicado através do envio de uma mensagem *prepare*. Cada réplica aguarda um quórum de $2f$ mensagens *prepare*. A partir deste momento, sabe-se que todas as réplicas corretas concordaram com o número de sequência n na visão v para a mensagem m.

Para assegurar o ordenamento das mensagens entre visões, a terceira fase de $commit$ é necessária. Nesta fase, cada réplica i realiza a difusão da mensagem *commit* informando as outras réplicas que tanto a mensagem de *pre-prepare* quanto o quórum contendo $2f$ mensagens de *prepare* relacionadas a mensagem m na visão v foram aceitas. Ao receber um quórum de $2f + 1$ mensagens *commit* de réplicas distintas, a requisição é dita *committed*. Após uma requisição atingir o estado *committed*, assegura-se que a informação acerca do seu número de sequência não se perderá durante o protocolo de troca de visão e, portanto, pode ser executada. Uma requisição *commited* r é executada pelas réplicas após todas as requisições com número de sequência menor do que o número de sequência de r forem executadas.

Três otimizações são implementadas visando diminuir os passos de comunicação:

1. **Execução por tentativa:** Permite que processos executem uma requisição logo após esta estar preparada desde que as requisições anteriores estejam confirmadas. Através desta técnica pode-se eliminar a fase de *commit* pois a resposta de uma requisição preparada é retornada ao cliente cliente enquanto as mensagens de *commit* podem ser enviadas juntamente com as próximas mensagens enviadas pela réplica (através de técnicas de *piggybacking*). No caso de uma troca de visão, as mensagens não confirmadas podem ser abortadas e o estado restaurado ao último *checkpoint* válido no sistema.

2. **Requisições de leitura:** Requisições de operações que não modificam o estado do serviço podem ser executadas imediatamente após o seu recebimento.

3. **Agrupamento de requisições:** Ao invés de realizar a ordenação das operações uma a uma, o protocolo aguarda um número pré-determinado de operações para serem sequenciadas em uma única vez. Desta forma,

o número de vezes que o protocolo de acordo é executado é reduzido e o sistema se torna mais eficiente quando está sob grande carga.

O primário de uma dada visão v pode ser faltoso, evitando que o protocolo progrida, por exemplo, através da omissão de mensagens contendo as propostas de número de sequência. Ao detectar que requisições pendentes não foram executadas após um dado intervalo de tempo, uma réplica passa para uma nova visão $v + 1$ e envia uma mensagem *view-change* a todos os processos informando sua decisão, juntamente com os seus conjuntos $\mathcal{P}$ e $\mathcal{Q}$. Os conjunto $\mathcal{P}$ e $\mathcal{Q}$ possuem todas as mensagens *pre-prepare* e *prepare*, respectivamente, de visões anteriores. Após esta mensagem ser emitida, a réplica deixa de participar da visão, ignorando quaisquer mensagens exceto as mensagens de *checkpoint*, *view-change* e *new-view*.

As réplicas aceitam uma mensagem *view-change* apenas se as mensagens em $\mathcal{P}$ e $\mathcal{Q}$ pertencem a visões menores que $v + 1$. Uma vez aceita, a réplica envia uma mensagem *view-change-ack* ao primário da visão $v + 1$. Então, o novo primário p da visão $v+1$ monta um certificado de troca de visão composto por uma mensagem *view-change* e $2f - 1$ mensagens *view-change-ack* recebidas durante o processo de troca de visão. Tal certificado atesta que a maioria das réplicas corretas concordou com a troca de visão, além do que, permite a criação de um *checkpoint* tornando possível a definição de números de seqüência para operações pendentes da visão anterior contidas nos conjuntos $\mathcal{P}$ e $\mathcal{Q}$. Por último, o novo primário envia uma mensagem *new-view* às replicas contendo o *checkpoint* e as mensagens pendentes ordenadas. Réplicas aceitam e instalam a nova visão desde que tenham aceitado a mensagem *view-change* correspondente.

O histórico de mensagens necessário para validar certificados pode crescer indefinidamente. Para evitar que isto ocorra, um mecanismo de coleção de lixo é utilizado. O algoritmo tenta realizar a limpeza de dados antigos toda vez que o número de sequência da última operação executada atinge um valor divisível por K. Este período é chamado de período do *checkpoint*. Quando existe uma prova de que o estado está correto em um *checkpoint* diz-se que este é estável. Cada réplica mantém além do estado atual, o estado do último *checkpoint* estável e os estados de todos os *checkpoint* não-estáveis posteriores ao *checkpoint* estável. No momento em que o *checkpoint* não-estável tirado após a execução da operação com número de sequência n se torna estável, o histórico de todas as mensagens referentes a requisições com número de sequência menor do que n pode ser removida. Além disso, a marca d'água inferior é aumentada para que mensagens em transito referentes a requisições antigas sejam ignoradas. Se uma réplica está atrasada em relação ao último *checkpoint* estável, ela assume o estado de tal *checkpoint* como seu estado atual.

3.2.2 Zyzzyva

Zyzzyva [57], a exemplo do BFT, é um protocolo de replicação de máquinas de estados. Seu objetivo é reduzir o custo e simplificar o projeto de sistemas tolerantes a faltas bizantinas utilizando tal abordagem. Para atingir este objetivo, Zyzzyva utiliza técnicas especulativas.

Zyzzyva adota o modelo tradicional de máquina de estados no qual um primário propõe números de ordem para as requisições realizadas pelos clientes. Entretanto, Zyzzyva não executa um protocolo de acordo que defina uma ordem final para a execução, mas executa de forma especulativa. Logo, o estado de réplicas corretas distintas pode divergir, fazendo com que clientes recebam respostas diferentes. Entretanto, a aplicação não toma conhecimento destas divergencias, vendo uma máquina de estados replicada que fornece a mesma propriedade de linearização [59] encontrada em PBFT. Para que isto seja possível, clientes participam ativamente do protocolo. Parte do histórico das computações realizadas são enviadas na resposta, permitindo que clientes identifiquem respostas e históricos estáveis. Caso uma resposta especulativa e seu histórico estejam estáveis, ela pode ser utilizada pelo cliente.

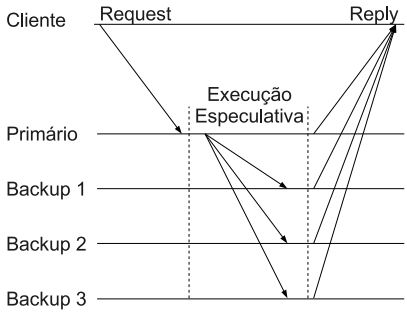
Assim sendo, Zyzzyva deve assegurar que respostas se tornem estáveis. Réplicas são responsáveis por assegurar que isto ocorra. Entretanto, clientes podem prover mais informações de forma a acelerar o processo ou tornando a resposta estável, ou causando uma troca de visão. Devido a esta abordagem, clientes podem atuar sobre as respostas em uma ou duas fases ao invés das três fases costumeiramente necessárias [60, 61].

Zyzzyva é composto por três sub-protocolos: acordo, mudança de visão e *checkpoint*. O protocolo de acordo ordena as requisições feitas pelos clientes para execução pelas réplicas, o protocolo de mudança de visão coordena a eleição de um novo primário quando o primário atual é faltoso ou o sistema está muito lento, por fim, o protocolo de *checkpoint* permite que mensagens antigas possam ser eliminadas e reduz o custo de implementação do protocolo de mudança de visão.

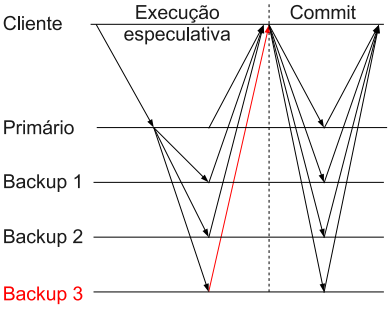
O protocolo é executado por $3f + 1$ réplicas, e sua execução é organizada em uma sequência de visões. Cada visão possui um único primário responsável por liderar o sub-protocolo de acordo.

O algoritmo funciona da seguinte forma: o cliente envia uma requisição ao primário. Este encaminha a requisição e o seu número de sequência às réplicas, que a executam e enviam ao cliente a resposta da execução da operação requisitada. Três possibilidades surgem neste momento: caso o cliente receba $3f + 1$ respostas consistentes, isto é, possuindo mesmo resultado e histórico, então o cliente considera que a requisição foi completada com sucesso, aceita a resposta como válida e a entrega a aplicação (figura 3.3(a)). Caso o cliente receba entre $2f + 1$ e $3f$ mensagens consistentes, ele monta um certificado

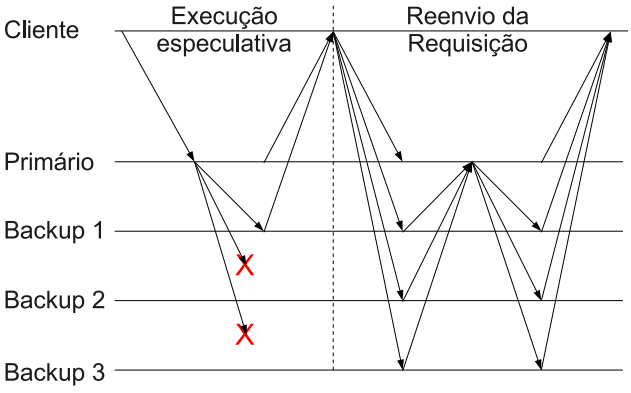
de *commit* composto por $2f + 1$ respostas consistentes e o envia às réplicas. Uma vez que $2f + 1$ réplicas enviem uma resposta confirmando o recebimento do certificado, o cliente considera a requisição completa e repassa o resultado a aplicação (figura 3.3(b)). Finalmente, o cliente pode receber menos de $2f + 1$ respostas consistentes. Nesta situação a requisição é enviada novamente a todas as réplicas, que a encaminham ao primário para que um número de sequência seja atribuída e a operação acabe sendo executada (figura 3.3(c)). Caso cliente detecte sinais de má-conduta por parte do primário, como número de sequência distintos para uma mesma requisição, o cliente envia uma mensagem às replicas junto a prova de má-conduta causando o início do protocolo de troca de visão.



(a) Com $3f + 1$ respostas consistentes



(b) Com $2f + 1$ respostas consistentes



(c) Com menos de $2f + 1$

Figura 3.3: Troca de mensagens durante a execução normal do PBFT.

O sub-protocolo de troca de visão é responsável por coordenar a eleição de um novo primário. Isto ocorre caso o sistema esteja lento ou caso o primário seja faltoso. Tal protocolo deve assegurar que nenhuma mudança no histórico que afete uma requisição aceita pelo cliente irá ocorrer. O sub-protocolo de troca de visão funciona da seguinte forma: uma réplica correta que suspeita do primário da visão v envia uma mensagem *I-hate-the-primary* para todas as réplicas exprimindo sua falta de confiança no primário da visão v. Caso uma réplica i receba $f + 1$ mensagens *I-hate-the-primary* na visão v, então i se engaja na mudança de visão enviando uma mensagem *view-change*, contendo um certificado composto por $f + 1$ mensagens *I-hate-the-primary*, e ignorando qualquer mensagem exceto as de *checkpoint*, *view-change* e *new-view*. A partir deste momento, o protocolo segue como o protocolo BFT, isto é, o primário da visão v coleta $f + 1$ mensagens *view-change*, calcula um conjunto de mensagens que possui um número de sequência na visão anterior e envia mensagem *new-view* a todas as réplicas completando a troca de visão. Há uma mudança na forma como o conjunto de mensagens e os números de sequência são atribuídos pelo primário da nova visão em relação ao protocolo BFT. Detalhes são discutidos em [57].

O sub-protocolo de *checkpoint* permite que réplicas construam pontos de sincronização. De forma semelhante ao protocolo BFT, cada réplica mantém além do seu próprio estado o estado da réplica no momento em que o protocolo de *checkpoint* determinou. Este estado *checkpoint* permite que réplicas atrasadas sejam trazidas a um estado mais avançado, permite o descarte de mensagens e estado relativos a mensagens antigas e auxilia o protocolo de mudança de visão, permitindo que apenas operações ocorridas após o último *checkpoint* estável seja processada.

3.3 Modelos de Faltas Híbridos

O presente trabalho adota um modelo de faltas híbrido, sendo que parte do sistema é visto como sendo confiável e sujeito apenas a faltas de *crash* enquanto outra parte do sistema opera no modelo de faltas bizantina. Com este tipo de modelo é possível criar algoritmos mais eficientes do que quando se está sob um modelo de falta totalmente bizantino. Pode-se observar o modelo de falta híbrido como sendo um modelo de faltas no qual um modelo de sistema bizantino clássico foi acrescido de um componente confiável capaz de fornecer serviços que, por construção, não podem ser subvertidos. Sob esta perspectiva, tal modelo se aproxima de modelos adotados em sistemas assíncronos com falta de *crash* acrescidos de um detector de falhas. Isto porque ambos os modelos, um componente que age como uma caixa preta fornece serviços de forma a simplificar o algoritmo, encapsulando algum aspecto em forma de serviços. No caso de detectores de falhas, aspectos de sincronia são

encapsulados. Com isso, é possível, por exemplo, executar as partes cruciais de um protocolo na parte do sistema que assume apenas a ocorrência de faltas de *crash*, tornando assim o protocolos BFT mais simples.

No que segue, são abordados alguns trabalhos na literatura que utilizam modelos acrescidos de componentes confiáveis capazes de fornecer serviços tolerantes a faltas bizantinas.

3.3.1 TTCB

Trusted Timely Computing Base (TTCB) [20] é um componente confiável capazes de prover serviços normalmente não encontrados em ambientes convencionais. TTCB é um tipo de *wormhole*, *wormholes* possibilitam a hibridização do modelo de sistema, por exemplo, parte do sistema está sujeito a faltas bizantinas enquanto outras partes estão sujeitos a faltas de crash.

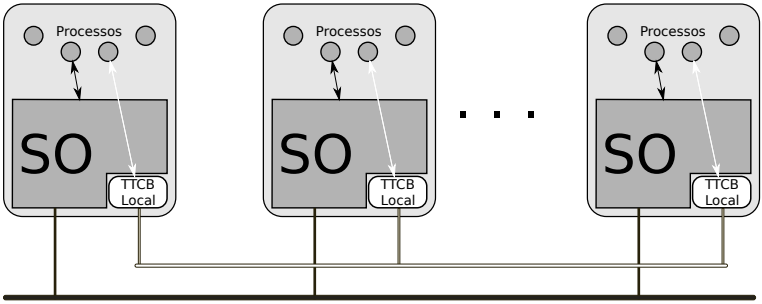


Figura 3.4: Arquitetura de um sistema utilizando TTCB.

TTCB apresenta características de tempo-real, é seguro e está sujeito apenas a faltas de crash. Entretanto, a aplicação que utiliza os serviços do TTCB podem estar sob um modelo qualquer, inclusive sujeitos a faltas bizantinas. Inclusive a utilização dos serviços fornecidos pelo TTCB pela aplicação pode se dar de forma abritária, apesar de que a especificação do TTCB ser, por construção, seja inviolável.

A arquitetura do sistema (figura 3.4) pode ser vista como uma aplicação tolerante a faltas bizantinas clássicas acrescido de um *wormhole* TTCB. Cada nó na rede possui um TTCB local, sendo este um pequeno componente conceitualmente separado e protegido do restante dos processos e sistema operacional do sistema. Os TTCBs locais estão interconectados por um canal de controle que assume-se ser seguro e síncrono. Os TTCBs locais em conjunto com os canais de controle compõe o TTCB. O restante do sistema é composto de processos usuais sujeitos a faltas bizantinas interconectados por uma rede ponto-a-ponto confiável.

O TTCB fornece os seguintes serviços aos nós do sistema:

1. **Serviço de autenticação local**: Fornece chaves criptográficas utilizadas pelos processos para se comunicar com o TTCB. Todo e qualquer processo que deseja utilizar o TTCB necessita autenticar e obter esta chave criptográfica.

2. **Serviço confiável de estampilha global**: Fornece estampilhas globais significativas. Tais estambilhas podem ser verificadas, pois TTCBs possuem relógios sincronizados.

3. **Serviço confiável de acordo de bloco**: Este serviço fornece uma primitiva de acordo para a aplicação implementada na rede síncrona de controle do TTCB. Este acordo possui como restrição o tamanho dos valores propostas, vinte bytes. Três primitivas compõe o serviço: TTCB_propose, TTCB_decide, decision. A proposta de um processo é feita através da primitiva TTCB_propose, o resultado de um acordo é obtido através da primitiva TTCB_decide. $decision$ é uma função aplicada ao conjunto de propostas que definem o valor acordado.

As primitivas que compõe o serviço confiável de acordo de bloco do TTCB satisfazem as seguintes propriedades:

1. **TBA1 Termination**: Todo processo correto acabará por decidir um valor.

2. **TBA2 Integrity**: Todo processo correto decide no máximo uma vez.

3. **TBA3 Agreement**: Dois processos corretos não decidem valores diferentes.

4. **TBA4 Validity**: Se um processo correto decide um valor, então o resultado é obtido pela aplicação da função *decision* ao valor acordado.

5. **TBA5 Timeliness**: Dado um instante $tstart$ e uma constante $TTBA$, o resultado do acordo estará disponível no TTCB no instante $tstart + TTBA$.

As primitivas TTCB_propose e TTCB_decide necessitam de uma série de argumentos, e retornam um conjunto de valores. A primitiva TTCB_propose possui necessita de: a identidade do processo perante o TTCB (eid), lista com a identidade dos processos que participarão do acordo($elist$), estampilha indicando o instante em que as propostas não serão mais aceitas ($tstart$), uma função que será aplicada ao valor acordado($decision$) e, por fim, a proposta do processo $value$. Como resultado da execução desta primitiva, o processo

Algorithm 2 Algoritmo de consenso basedo no TTCB

```
1: procedure CONSENSUS(elist, tstart, value)
2:     hash − v ←⊥
3:     bag ← ∅
4:     round ← 0
5:     phase ← 1
6:     send (elist, tstart, value) to ∀p : p ∈ elist

7:     loop
8:         repeat
9:             if phase = 2 then
10:                coord = (round mod n)
11:                value ← {M.value : M = NEXTSENDERMSG(coord, elist, bag)
12:            end if
13:            out_prop ← TTCB_propose(eid, elist, tstart, TBA_MAJORITY,
    HASH(value))
14:            repeat
15:                out_dec ← TTCB_decide(out_prop.tag)
16:            until out_dec.error ≠ TBA_RUNNING
17:            tstart ← tstart + Tfunc(α, round)
18:            round ← round + 1
19:            if (2f + 1 processos propuseram) ∧ (menos de f + 1 processos propuseram o
    mesmo valor) then
20:                phase ← 2
21:            end if
22:        until f + 1 processos propuseram o mesmo valor
23:    end loop

24:    on receive msg from p
25:        bag ← bag ∪ msg

26:    when (hash − v ≠⊥) ∧ (∃M ∈ bag : HASH(M.value) = hash − v)
27:        if phase = 2 then
28:            send M to ∀p : p ∈ (elist − processos que propuseram HASH(M.value))
29:        end if
30:        decide M.value
31: end procedure
```

irá receber um identificador da instância de acordo (tag) e um código de erro ($error$).

De forma análoga, a primitiva TTCB_decide necessida apenas de um identificador caracterizando a instância de acordo (tag) a qual se está referindo. Como resultado de tal primitiva obtém-se, além do resultado do protocolo de acordo $value$, uma máscara de bits indicando processos que propuseram $value$ ($proposed - ok$), uma máscara de bits indicando processos que propuseram algum valor dentro do limite de tempo ($proposed - any$) e um código de erro ($error$)

O protocolo de consenso baseado no TTCB proposto em [5] (algoritmo 2) é capaz de tolerar até f faltas com $n = 3f+1$ processos. O algoritmo opera em duas fases. Durante a primeira fase, processos propõem através do TTCB o *hash* de suas propostas até que $f + 1$ processos possuam o mesmo *hash*. A segunda fase do protocolo é apenas utilizada quando não se conseguiu atingir $f+1$ processos com mesmo *hash*. Neste caso, o paradigma de coordenador rotativo é adotado. O algoritmo evolui em rodadas assíncronas, sendo que cada rodada possui um coordenador. O coordenador realiza sua proposta através do TTCB. No momento em que $f + 1$ processos corretos dão o seu aval à proposta feita pelo coordenador, o consenso foi atingido e a proposta é assumida como o resultado do consenso.

Este algoritmo é o primeiro a adotar TTCB para resolver consenso em um modelo de faltas de bizantino. Em execuções sem falha, este algoritmo possui a vantagem de atingir acordo em apenas dois passos de comunicação, quando os processos corretos possuem mesma proposta inicial, e quatros passos em outros cenários. O TTCB foi proposto previamente[20] mas foi utilizado para realizar o *reliable multicast* de mensagens em um ambiente assíncrono com um número qualquer de processos faltosos.

3.3.2 A2M

Em um sistema distribuído composto apenas por elementos não confiáveis existe limites teóricos em relação ao número mínimo de nós necessários para a execução de protocolos distribuídos. Por exemplo, o problema de acordo bizantino [26], replicação de máquinas de estados [36] e *atomic broadcast* [2] exigem que $f \leq \lfloor \frac{n-1}{3} \rfloor$ dentre as n réplicas sejam faltosas.

Attested Append-Only Memory (A2M) [4] é um componente confiável que provê aos nós um serviço confiável de *log* pequeno, de fácil implementação e passível de verificação formal de corretude. Por meio deste mecanismo confiável, pode-se implementar uma versão do algoritmo PBFT [36] capaz de tolerar $f \leq \lfloor \frac{n-1}{2} \rfloor$ réplicas faltosas.

A redução no número de processos necessários pelo protocolo PBFT conseguido através do uso do A2M se deve a impossibilidade de que um processo faltoso envie mensagens distintas para outros processos, quando a mesma mensagem deveria ter sido recebida. A2M permite que processos detectem tal comportamento por parte de processos maliciosos através de um serviço de *log*.

A2M provê ao nó da rede um conjunto de *logs* ordenados, inquestionáveis e confiáveis. Cada um destes *logs* possui um identificador q (único em um mesmo nó) que consiste de uma sequência de valores. Cada valor possui: (1) um número de sequência crescente por log, e (2) um *hash* criptográfico feito a partir de todas as entradas do *log* até a própria entrada. Apenas um su-

fixo do *log* é armazenado pelo A2M, compreendendo todos os valores desde a posição $\mathcal{L} \geq 0$ até a posição $\mathcal{M} \geq \mathcal{L}$.

A seguinte interface é oferecida pelo A2M:

- `append`(q, x): incrementa $\mathcal{H}$ em 1, inclui o valor x ao fim do *log* identificado por q (posição recém criada pelo avanço de $\mathcal{H}$), e calcula o *hash* criptográfico correspondente. Caso não seja possível a inclusão no *log* a operação falha.

- `lookup`(q, n, z): obtém um atestado de *lookup* referente a posição n do *log* q, sendo z é um nonce. Existem quatro possibilidades: (1) caso $n > \mathcal{H}$ é retonado um atestado *unassigned*. (2) Caso $n < \mathcal{L}$ é retornado um atestado *forgotten*. (3) Caso a posição tenha sido pulada através do procedimento *advance* é retornado um atestado *skip*. E por fim, (4) n está associado a um valor v, fazendo com que um atestado *assigned* contendo o *hash* e v sejam emitidos.

- `end`(q, z): possui o mesmo efeito do método *lookup* quando operado na última posição preenchida do *log*. A diferença reside no fato de ser possível inferir, a partir do atestado, que a posição n era a última do *log* no momento em que tal procedimento foi invocado.

- `truncate`(q, n): elimina todas as entradas com número de sequência menor que n, fazendo $\mathcal{L} \leftarrow n$.

- `advance`(q, n, d, x): permite que o *log* q pule para a posição n fazendo $\mathcal{H} \leftarrow n$. O valor d é utilizado pelo A2M no cálculo do *hash*.

Diversas abordagens para a implementação do A2M são propostas. O autor propõe as seguintes alternativas: A2M pode ser um componente de hardware especializado, um serviço provido por um monitor de máquina virtual, um serviço provido à uma máquina virtual por uma segunda máquina virtual confiável, um processo confiável rodando sobre o mesmo sistema operacional que a aplicação e, por fim, um serviço confiável oferecido por uma máquina dentro de um sistema distribuído.

3.3.3 TrInc

Através da inclusão de um componente confiável, tal como A2M [4] e TTCB [5, 20], em um modelo de sistema assíncrono sob o modelo de faltas bizantino, pode-se evitar equivocação (*equivocation*). Isto é, um processo malicioso não é capaz de enviar uma mensagem a um processo e uma mensagem adulterada a outro. Ao se impedir que processos faltosos realizem este procedimento, problemas que em um sistema bizantino clássico necessitam de $n \geq 3f + 1$ para tolerar f processos faltosos podem ser resolvidos com

um número menor de processos, aumentando a resiliência. Por exemplo, apresenta em [4] um algoritmo de replicação de máquinas de estado capaz de tolerar $f = \lfloor \frac{n-1}{3} \rfloor$ processos faltosos com n réplicas.

[62] apontam que um dos grandes problemas de sistemas baseados em componentes confiáveis é a necessidade de garantir a inviolabilidade de tais componentes. Além disso, A2M necessita de uma quantidade razoavel de memória e apresenta uma série de serviços bastante sofisticados, introduzindo tamanha complexidade que pode-se não estar disposto a tomar como hipótese a inviolabilidade de tal sistema em um ambiente hostil.

TrInc [62] é um componente confiável menor e com uma semântica mais simples do que o A2M. Isto facilita a implantação em um sistema distribuído real, pois é possível implementá-lo a partir de hardware confiável já existente e disponível, o TPM [63, 64]. TrInc possui ainda um modo de operação que dispensa o uso de criptografia assimétrica, necessária no A2M, acarretando uma diminuição no processamento criptográfico necessário para implementar o protocolo.

Para utilizar o TrInc, cada usuário deve possuir um *hardware* especializado chamado *trinklet* em seu computador e um canal de comunicação não-confiável com este hardware. Para atestar uma mensagem m, deve-se gerar um atestado a partir do *trinklet*. O *trinklet* associa m ao valor de um contador interno e assegura que nenhuma outra mensagem será associada a tal valor. *Trinklet* utiliza um contador monotonico que é incrementado a cada atestado emitido.

Cada *trinklet* possui uma identidade única *I* e um par de chaves pública e privada, K_{pub} e K_{priv} respectivamente, e um atestado $\mathcal{A}$ que prova que os valores *I* e K_{pub} pertencem a um *trinklet* válido. As seguintes operações são fornecidas por um *trinklet*:

1. `Attest`: recebe três parâmetros: a identidade do contador a ser utilizado i, o novo valor para o contador c' e o *hash* h da mensagem m que será atestada. Um atestado assinado contendo os valores I, i', c', c e h é emitido, onde c é o valor do contador antes de fazer o contador do trinklet assumir o valor c'. Atestado é emitido apenas caso $c \leq c'$

2. `CreateCounter`: cria um novo contador e retorna o seu identificador i.

3. `FreeCounter`: Se i é a identidade de um contador válido, o deleta.

4. `GetCertificate`: Retorna um certificado da validade do *trinklet*: $\langle I, K_{pub}, \mathcal{A} \rangle$

Um processo que receba um atestado emitido por um *trinklet* é capaz de verificar a sua autenticidade. Para isto, o certificado `GetCertificate`

deve ser conhecido e válido pelo processo que recebeu o atestado, permitindo acesso à chave publica K_{pub} do *trinklet* responsável pela emissão do atestado em questão.

Um algoritmo que implementa o A2M utilizando o TrInc é apresentado em [62]. Isto permite que qualquer algoritmo já desenvolvido para o A2M pode ser implementado com o TrInc. Exemplos de tais algoritmos são: PBFT [36], Q/U [65], SUNDR [66].

3.3.4 VM-FIT

VM-FIT [8] faz uso da tecnologia de virtualização para prover serviços tolerantes a faltas bizantinas e intrusão. A tecnologia de virtualização provê um componente chamado *hypervisor* que está totalmente isolado do sistema operacional convidado e fornece a real implementação de diversos serviços. Através do *hypervisor* é possível prover um componente confiável isolado que não possui todas as vulnerabilidades do sistema operacional convidado que está rodando o serviço. Além disso, o *hypervisor* tem total controle sobre o sistema operacional convidado, portanto, pode oferecer suporte a recuperação pro-ativa do sistema operacional convidado de forma eficiente.

A arquitetura do VM-FIT fornece um suporte genérico para replicação de serviço disponibilizados em rede. A comunicação remota é interceptada, de forma transparente, no nível do *hypervisor*, e provê suporte a recuperação pró-ativa. Neste trabalho as seguintes premissas são assumidas:

- Clientes utilizam uma interação de acesso ao serviço remoto baseado em requisição-resposta.
- Os serviços possuem comportamento determinístico
- Faltas bizantinas podem ocorrer em um número limitado de réplicas do serviço.

O VM-FIT é um sistema de replicação de máquinas de estado clássico. Entretanto, cada réplica não é composta apenas por um sistema operacional rodando o serviço. Cada réplica é composta por um *hypervisor* rodando diretamente sobre o hardware. Sobre este hypervisor encontram-se duas máquinas virtuais. A primeira máquina virtual, chamada *Domínio NV*, é confiável. A segunda máquina virtual é responsável por rodar o serviço replicado.

O padrão de comunicação difere um pouco do modelo clássico, no qual clientes e réplicas se comunicam diretamente. No VM-FIT (figura 3.5), o *hypervisor* captura as requisições realizadas por um determinado cliente e direcionando-os ao *Domínio NV*. O *Domínio NV* dissemina esta requisição para o *Domínio NV* encontrado nas outras réplicas e em seguida a repassa para a máquina virtual que está rodando o serviço propriamente dito. Todas

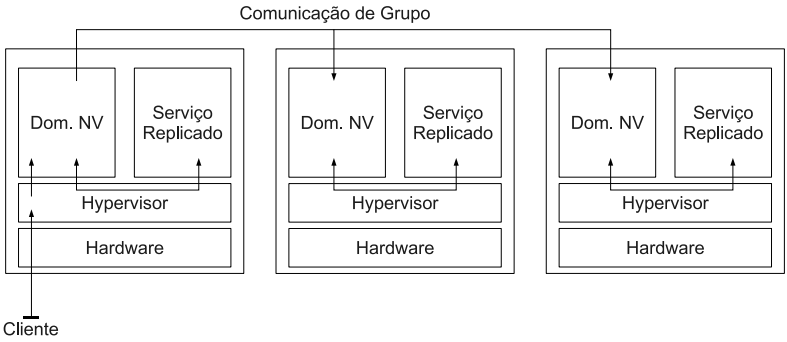


Figura 3.5: Arquitetura do sistema VM-FIT.

as réplicas executam a requisição realizada pelo cliente. A resposta enviada pela réplica é interceptada pelo *hypervisor* que a redireciona ao *Domínio NV*. Em posse do resultado da execução da sua réplica, o *Domínio NV* notifica as demais de tal resultado, e aguarda que $f + 1$ respostas idênticas sejam colecionadas que por sua vez, através de primitivas de comunicação de grupo, a dissemina entre todos os *Domínio NV*. Ao colecionar $f + 1$ respostas idênticas a uma dada requisição, o *Domínio NV* envia a resposta ao cliente.

3.4 LBFT

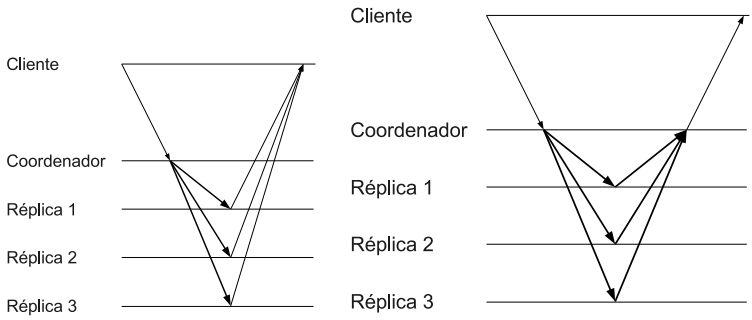
em [6] propõe o uso de técnicas de replicação de máquinas de estado utilizando-se apenas uma máquina física, através da tecnologia de virtualização. Desta forma, é possível utilizar de forma mais eficiente o grande poder computacional encontrado nos computadores atuais, muitas vezes subutilizados, para fornecer tolerância a intrusão, e a erros de projeto que possam a vir fazer o serviço replicado se comportar de forma inesperado, gerando respostas errôneas.

Um dos desafios fundamentais em aumentar a confiabilidade neste cenário é fazer com que réplicas do serviço falhem independentemente. Por exemplo, se exatamente o mesmo *bug* pode fazer com que um atacante malicioso tome controle de todas as réplicas, replicação não será uma ferramenta útil. Uma forma de se fazer isso é através de técnicas de diversidade do sistema [67]. Esta técnica dificultam a exploração de falhas de software através da heterogeneidade do ambiente.

Duas abordagens diferentes são propostas: LBFT1 e LBFT2. No LBFT1 (figura 3.6(a)) toma-se como premissa a existência de um coordenador confiável e um sistema assíncrono, sendo que a replicação se da de forma

não-transparente, isto é, os clientes estão cientes da existência da replicação. Em uma mesma máquina são colocadas $2f + 1$ réplicas de execução mais o coordenador, sendo toleradas até f faltas. O protocolo é simples: ao receber a requisição de uma operação oriunda do cliente, o coordenador atribui um número de sequência a esta operação e realiza uma difusão da operação e número de sequência para as réplicas de execução. As réplicas então executam as operações requisitadas em ordem crescente de número de sequência e enviam o resultado ao cliente, que irá esperar uma maioria ($f+1$) de respostas idênticas para assumir o resultado como sendo correto.

LBFT2 (figura 3.6(b)) utiliza um coordenador confiável, sistema assíncrono, mas difere do LBFT1 pois provê replicação transparente. O cliente está ignorante do fato que existe replicação no serviço que está utilizando. Para isto, uma mudança no protocolo é realizada. O protocolo funciona da seguinte forma: o cliente envia ao coordenador uma requisição para a execução de uma dada operação. O coordenador atribui a esta requisição um número de sequência e a repassa às réplicas de execução, que executam as operações em ordem crescente de número de sequência. Em posse do resultado da operação, a réplica não mais envia requisição ao cliente, mas ao coordenador. Este realiza a votação das respostas, escolhendo o resultado da maioria, e responde ao cliente. Desta forma, o cliente não necessita realizar nenhuma espécie de votação, utilizando o sistema réplicado como se não o fosse.



(a) LBFT1 (b) LBFT2

Figura 3.6: Troca de mensagens no LBFT. Linhas grossas indicam comunicação dentro da mesma máquina física.

3.5 Servidor de Consenso no Modelo de *Crash*

Protocolos de acordo são a base para a resolução de vários problemas distribuídos, tais como *atomic commit*, *group membership*, *total order broadcast*, e realizam papel fundamental em aplicações distribuídas como aplicações transacionais.

O servidor de consenso [11] é um *framework* que faz uso de um serviço de consenso genérico para construção de protocolos tolerantes a faltas de *crash*. O serviço de consenso é prestado por n_c servidores de consenso, que se encontram diretamente relacionados a resiliência desejada, isto é, quanto maior o número de servidores de consenso mais resiliente é o sistema.

Para que o serviço de acordo seja adaptado para diferentes situações e problemas, é utilizado um filtro de consenso. Cada acordo é ocorre na forma de uma iteração cliente-servidor. Clientes enviam propostas aos servidores de acordo que, por sua vez, respondem com o valor acordado. Existe uma diferença entre protocolos cliente-servidor usuais: a multiplicidade. Ao invés de um relacionamento de um para um tradicional, há um relacionamente $n_c - n_s$ onde n_c é o número de clientes e n_s o número de servidores.

O uso de serviços genéricos para a construção de sistemas distribuídos ou serviços de mais alto-nível se tornou comum. Diversos exemplos podem ser encontrados como: servidores de tempo, de arquivos, de nome, de autenticação, entre outros. Apesar disto, poucas propostas foram feitas para serviços dedicados à construção de procolos de acordo tolerantes a faltas, como broadcast atômico. Estes protocolos, são projetados isoladamente sem que compartilhem uma infra-estrutura comum. A exceção a regra é o serviço de *group membership*, utilizado para implementar diversos protocolos de ordem total.

sugerem o uso de consenso como paradigma central para a construção de sistemas distribuídos confiáveis [1]. O serviço de consenso mostra como transformar diversos problemas de acordo em consenso de forma sistemática.

O sistema adota uma arquitetura em camadas. A primeira camada trata as falhas de comunicação e detecção de maquinas faltosas (detectores de falhas [22]), a segunda camada entre o serviço de consenso genérico responsável pelo protocolo de consenso, e sobre este estão os problemas de acordo específicos.

Além do *reliable multicast* (descrito na seção 2.2.4 do capítulo 2), é utilizada a primitiva de comunicação *multisend*. Tal primitiva de comunicação, ao contrário do *reliable multicast*, fornece garantias acerca da entrega de mensagens apenas caso o emissor seja correto. Desta forma, é possível o recebimento de uma dada mensagem apenas por parte dos processos correto que compõe o sistema distribuído. A implementação do *multisend* é bastante simples consistindo apenas no envio da mensagem a todos os processos (algo-

ritmo 3).

Algorithm 3 Algoritmo de *multisend*

```
1: procedure MULTISEND(msg, group)
2:     for ∀p : p ∈ group do
3:         send msg to group
4:     end for
5: end procedure
```

Os processos são classificados em papéis dependendo de sua função no sistema. Existem três papéis distintos no serviço de consenso:

1. **Iniciador**: inicia o acordo entre um subconjunto de clientes.
2. **Clientes**: são processos que irão resolver um dado problema de acordo através do serviço de consenso.
3. **Servidor de consenso**: são os processos que provêm o serviço de consenso, sendo responsável pela execução do protocolo de consenso.

Um processo pode realizar mais de um papel no sistema, sendo que o papel de servidor de consenso pode ser realizado por uma fração/todos os clientes ou por máquinas especializadas. Por simplicidade, será considerado que cada papel será realizado por entidades diferentes.

O filtro de consenso é um componente anexado a todo processo servidor de consenso s_j, sendo definido por 2 parâmetros: um predicado `CallInitValue` e a função `InitValue`. O predicado `CallInitValue` indica quando o protocolo de consenso pode ser iniciado. Este predicado deve ser estável, isto é, uma vez que assuma valor verdadeiro permanece neste estado durante toda a execução do sistema distribuído. A função `InitValue` retorna o valor inicial, calculado sobre valores oriundos dos clientes, que deve ser utilizado pelos servidores ao iniciar o protocolo de consenso.

É dito que um filtro de consenso possui vivacidade em um determinado servidor de consenso se existir um instante no qual o predicado `CallInitValue` é verdadeiro e a função `InitValue` retornar um valor em um instante qualquer.

Duas propriedades são asseguradas pelos servidores de consenso:

1. **CS-Acordo**: dois clientes quaisquer não receberão mensagens de decisão diferentes para uma mesma instância de acordo.
2. **CS-Terminação**: se o filtro de consenso possuir vivacidade, então a mensagem informando o valor de decisão dos servidores acabará sendo recebida pelos clientes.

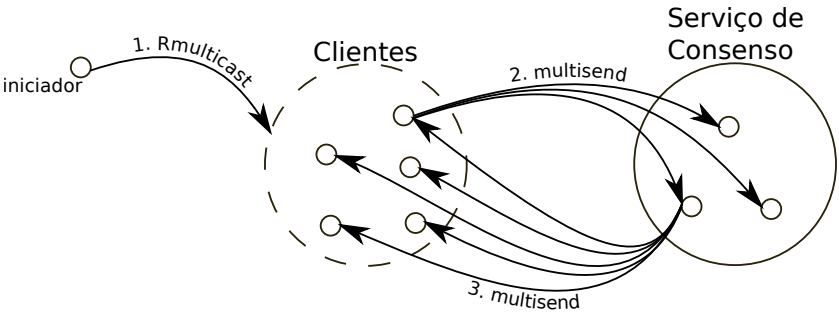


Figura 3.7: Padrão de comunicação no serviço de consenso. São representadas mensagens enviadas por um cliente e um servidor apenas por motivos de clareza.

O protocolo utilizado pelo serviço de consenso funciona da seguinte forma: o iniciador da partida a um acordo (algoritmo 4) realizando o *reliable multicast* de uma mensagem $\langle Cid, data, clients \rangle$ aos clientes que irão participar do mesmo. Como várias instâncias de acordo podem ocorrer durante a execução do sistema, um identificador Cid de indentifica unicamente cada uma. Ao iniciar uma instância de consenso, o iniciador pode enviar a todos os cliente um valor $data$ contendo informações acerca do acordo que deve ser realizado.

Algorithm 4 Iniciador no serviço de consenso de Guerraoui e Schiper.

1: **procedure** INITAGREEMENT($Cid, data, clients$)
2: **multisend** $\langle Cid, data, clients \rangle$ **to** $clientes$
3: **end procedure**

Ao receber uma mensagem oriunda dos iniciadores requisitando o início de um acordo, o cliente (algoritmo 5) calcula uma proposta $data'$ e a envia aos servidores de consenso através do *multisend* de $\langle Cid, data', clients \rangle$ e aguarda o resultado do consenso. Após a recepção do valor acordo, o cliente realiza qualquer computação que se fizer necessária para o problema de acordo específico que se está tratando.

Algorithm 5 Cliente no serviço de consenso de Guerraoui e Schiper.

1: **on *Rdelivery*** $Cid, data, clients$ **from** iniciador
2: $data' \leftarrow$ COMPUTEDATA($data$)
3: **multisend** $\langle Cid, data', clients \rangle$ **to** $servers$
4: **wait** reception of $\langle Cid, decision \rangle$ from any server

Os servidores de consenso (algoritmo 6) aguardam propostas da ins-

tância de acord Cid oriundas dos clientes até que o predicado do filtro de consenso `CallInitValue(`Cid`)` se torne verdadeiro. Neste momento, a função do cálculo de proposta do filtro de consenso `InitValue` é aplicada as mensagens recebidas até o momento, gerando uma proposta que resolve o problema de acordo requerido pelos clientes. Os servidores então executam um protocolo de consenso, elegendo uma das propostas calculadas como a solução para a instância Cid do acordo. Finalmente, tal proposta $decision$ é encaminhada aos clientes através do *multisend* da mensagem $\langle Cid, decision\rangle$.

Algorithm 6 Servidor no serviço de consenso de Guerraoui e Schiper.

Para cada instância Cid
1: **wait until** CALLINITVALUE(Cid)
2: $v \leftarrow$ INITVALUE($\{data'_i$: message$\langle Cid, data'_i, clients\rangle$ received from $c_i\}$)
3: $decision \leftarrow$ CONSENSUS(cid, v)
4: **multisend** $\langle Cid, decision\rangle$ **to** $clients$

3.6 Considerações Finais

Este capítulo apresentou alguns trabalhos considerados relevantes que influenciaram o desenvolvimento do Serviço Genérico de Consenso (SGC). Abordou-se técnicas de replicação ativa através da técnica de replicação de máquinas de estados, algoritmos de acordo, arquiteturas que adotam um modelos híbridos de faltas e o trabalho de Guerraoui e Schipper que introduziu o conceito de serviço de consenso.

O trabalho PBFT [36] foi o primeiro protocolo de replicação baseado em máquina de estados com a preocupação explicita em ser prático, acarretando em grande preocupação com a performance do sistema, o que trouxe um grande número de otimizações. Zyzzyva [57], por sua vez, veio a introduzir a noção de execução especulativa no contexto de faltas bizantinas com o intuíto de tornar o sistema mais eficiente em execuções sem falhas. Em cenários sem muitas faltas, Zyzzyva não incorre na penalidade da latência trazida pela execução de um protocolo de acordo. Entretanto, na presença de falhas dos processos ou rede (perdas e atrasos de mensagens) a participação do cliente se torna necessária no algoritmo, fazendo com que o número de trocas de mensagens aumente de dois para cinco. Além disso, os protocolos de *checkpoint* e troca de visão se tornam mais complexos necessitando de um maior número de passos de comunicação.

Nota-se que estes trabalhos buscam minimizar o custo da replicação, principalmente em situações nas quais o sistema não sofre faltas. Assim sendo, se incorre em penalidades na ocorrência de faltas em detrimento da eficiência de execuções sem faltas [68]. Além disso, ataques que visam a degradação de performance vem sendo documentados na literatura [69] e algumas soluções

para tais ataques foram propostos [58, 69, 70]. Para tornar sistemas tolerantes a faltas bizantinas realmente práticos, não basta assegurar a integridade dos dados, deve-se levar em consideração a perda de performance quando o sistema se encontra sob ataque.

Os algoritmos baseados em detectores de falhas desvinculam aspectos relativos ao *liveness* do sistema de aspectos de corretude. Assim, caracteristicas temporais e de sincronia não precisam ser tratados diretamente pelos algoritmos. Em um modelo de falta de *crash* tal separação se dá de maneira limpa e muito elegante, havendo uma total separação entre o algoritmo e o detector de falhas. Entretanto, faltas bizantinas não permitem uma separação tão explícita e o uso de detectores bizantinos (por exemplo, detectores de *muteness*[24] e *quietness*[25]) acabaram não sendo tão interessantes, já que há uma dependência mútua entre o algoritmo e o detector. Por esse motivo, em modelos bizantinos, é comum encontrar o tratamento de aspectos de *liveness* em conjunto com o restante do algoritmo, sem haver distinção trazida pelos detectores. O algoritmo de paxos [48] é um algoritmo de acordo que não utiliza detectores de falha e foi utilizado como base para o protocolo de acordo bizantino criado para o PBFT [36][1]. Estes algoritmos tiveram muita influência no desenvolvimento dos protocolos de acordo apresentados no capítulo 5.

Alguns modelos de sistema acrescidos de componentes confiáveis, caracterizando uma hibridização arquitetural do modelo de faltas, são descritos. Tais modelos inspiraram a adoção da tecnologia de virtualização como base para o desenvolvimento dos protocolos de servidores, permitindo uma melhora de performance através da redução do número de passos nos algoritmos de acordo e do aumento da resiliência do sistema. É possível imaginar a tecnologia de virtualização como sendo uma forma elegante para implementar técnicas de *wormhole* em software [71], devido a transparência que o sistema virtual oferece. Uma aplicação rodando sobre uma máquina virtual não sabe que está sobre um sistema virtualizado. Isto permite que serviços e componentes tolerantes a faltas de crash sejam implementados em partes protegidas do sistema virtualizado ou separado do sistema virtualizado.

[1] Uma descrição detalhada do protocolo de acordo utilizado no PBFT, isolada do sistema de replicação, pode ser encontrada em [3].

Capítulo 4

Servidor de consenso

O problema do serviço de consenso foi primeiramente proposto por Guerraoui e Schiper em [11], onde é sugerida a separação entre processos que necessitam solucionar um problema de acordo da execução de um algoritmo de consenso em um modelo de faltas *crash*.

O Serviço Genérico de Consenso (SGC) proposto aqui, estende o serviço de consenso de Gurraoui e Schiper para o modelo de faltas bizantinas. O principal objetivo do serviço de consenso é padronizar e modularizar a implementação de problemas de acordo. O SGC é exposto às aplicações na forma de um *framework*.

A redução de problemas como *atomic broadcast* para o consenso, em um ambiente bizantino, exige do mesmo uma propriedade de validade mais restritiva que a definição clássica usada na versão de *crash*. Esta necessidade é explicitada em [24]. Desta forma, adota-se a definição de validade de vetor para permitir a resolução de problemas de consenso. Assim sendo, o resultado obtido após a execução do protocolo de consenso é um vetor contendo propostas realizadas para a solução do problema de acordo que se deseja resolver.

Os processos que tomam parte no SGC são categorizados dependendo da função exercida por cada um no sistema. Cada função é representada por um papel, existindo três papéis no contexto do SGC: iniciador, cliente e servidor. O SGC é composto por n_s servidores, n_c clientes e n_i iniciadores. Assume-se que os servidores estão sendo executados em nós separados dos clientes e iniciadores, e os conjuntos de clientes e iniciadores podem apresentar intersecções.

Os processos iniciadores começam uma instância de consenso na qual clientes irão consultar os servidores. Processos clientes são aqueles que possuem um problema de acordo que necessita ser resolvido no contexto de um algoritmo de aplicação AP. Os servidores que suportam um serviço genérico de consenso são os responsáveis por coletar propostas dos clientes, criando e decidindo sobre um vetor válido de propostas a ser retornado aos clientes. O SGC é genérico e processos servidores são agnósticos aos significados das propostas e resultados. É de responsabilidade de cada cliente converter um

resultado recebido dos servidores em um valor útil para resolver o problema de acordo particular no qual os clientes estão interessados. Esta conversão é realizada através do conceito de filtro de consenso (FC). Este FC é composto por uma função determinística, chamada `Result`, que dado um vetor de decisão calculado pelos servidores, produz um valor que será consumido pelos clientes.

Assume-se que durante a execução de uma instância de consenso, existem no máximo f_s processos servidores faltosos, f_c processos clientes faltosos. Para suportar a presença de f_c clientes faltosos, o algoritmo requer que $n_c = 3f_c + 1$. O número de servidores necessários para suportar a presença de f_s servidores maliciosos está diretamente relacionado ao protocolo de acordo adotado pelos servidores. Caso o protocolo de consenso utilizado entre servidores exija $3f + 1$ processos para tolerar f processos faltosos, então serão necessários $3f + 1$ servidores. O SGC apresentado adota a tecnologia de máquinas virtuais e um modelo híbrido de faltas, permitindo que $n_s = 2f_s + 1$ processos sejam necessários para suportar até f_s servidores maliciosos. Os algoritmos de acordo são apresentados no capítulo 5. A arquitetura geral do SGC pode ser visualizada na figura 4.1.

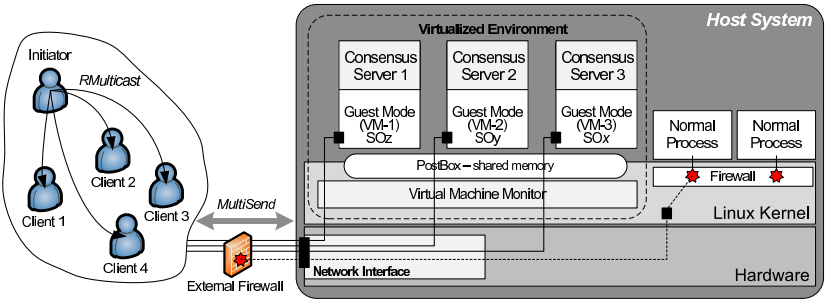


Figura 4.1: Arquitetura geral do Serviço Genérico de Consenso (SGC).

O conjunto de iniciadores pode ser composto por qualquer número de processos. Estes processos necessitam assegurar que todos os processos clientes corretos acabarão iniciando instâncias de consenso que devem ser resolvidas. Um iniciador malicioso pode realizar alguns ataques, entretanto, estes ataques são limitados pelo uso de um *multicast* confiável para o envio da mensagem de sincronização. O iniciador malicioso pode apenas tentar iniciar instâncias inválidas de consenso ou exaurir o espaço de valores de identificação de instâncias. O primeiro ataque é dependente de aplicação e deve ser tratado de acordo com o protocolo especifico de acordo. O último pode ser resolvido através de técnicas como *high and low water marks* [72].

De modo geral, o SGC opera da seguinte forma (figura 4.2): o iniciador dá partida a uma instância do consenso fazendo o `reliable multicast` de uma mensagem SYNC aos clientes. Como conseqüência, os clientes calculam e enviam suas propostas aos servidores através do *multisend*. Estas propostas são específicas ao problema de acordo que os clientes devem resolver em uma determinada aplicação. Os servidores coletam estas propostas, executam um protocolo de consenso e respondem aos clientes com o valor de um vetor acordado.

Dois modos de operação distintos podem ser adotados na relação entre iniciadores e clientes. No primeiro modo, assume-se que iniciadores e clientes são conjuntos distintos e a instância do consenso é iniciada através do *reliable multicast* de uma mensagem SYNC, como descrito assima. O segundo modo de operação dispensa o uso do *reliable multicast*. Para isso, exige-se que o conjunto de iniciadores e clientes coincidam. É responsabilidade do cliente e iniciador, neste segundo caso, assegurar que todos os processos corretos enviem suas propostas aos servidores em uma instância de consenso que deve ser resolvida. Caso esta condição não seja satisfeita, o SGC não garante que tal instância tenha um vetor atribuído.

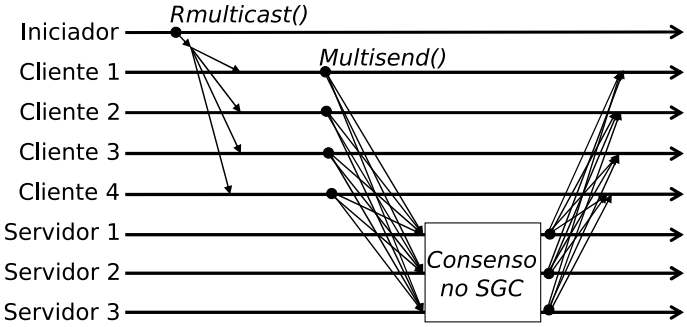


Figura 4.2: Padrão de comunicação de processos executando uma instância de consenso.

4.1 Base Algorítmica do SGC

Cada processo segue um dado algoritmo dependendo do papel que ele assume dentro do SGC. Os iniciadores seguem o algoritmo 7. Este algoritmo fornece um método chamado `RequireAgreement` para iniciar uma instância de consenso identificada pelo resultado da execução da função `CalculateIdentifier`, que deve ser implementada de acordo com as especificidades do problema a ser resolvido pelo SGC. Se o conjunto de clientes coincide com o conjunto de iniciadores, então o procedimento `RequireLo-`

`calStart` pode ser utilizado para iniciar um acordo. Este procedimento faz com que o cliente que o executou envie a sua proposta aos servidores. Especializações do SGC, ao utilizar este procedimento, devem se preocupar em garantir que todos os clientes corretos proponham valores de uma dada instância para que os servidores possam decidir.

Algorithm 7 Processo iniciador p_i do SGC

```
1: procedure REQUIRESTART
2:     idC ← CALCULATEID
3:     Rmulticast⟨SYNC; idC⟩ to clients
4: end procedure
5: procedure REQUIRELOCALSTART(idC)
6:     send ⟨SYNC; idC⟩ to p_i
7: end procedure
```

Os cliente do SGC seguem o algoritmo 8. Ao receber a mensagem SYNC do processo iniciador, o cliente calcula sua proposta e atribui a mesma à variável $proposal$ (linha 9). Esta proposição é então enviada aos processos servidores do SGC através de um `Multisend()`, juntamente com um identificador da instância de consenso `idC` e a assinatura correspondente (line 12). O cliente aguarda a chegada das respostas dos servidores. Quando o cliente acumular $f_s + 1$ respostas com o mesmo vetor decisão de servidores distintos (line 17), a função do filtro de consenso `Result` é aplicada ao vetor (linhas 18 e 19) e o controle é entregue a aplicação através da chamada do método `AgreementFinished`.

O processo servidor segue o algoritmo 9. Ao receber $n_c - f_c$ mensagens PROPOSE de clientes propondo um mesmo valor em uma mesma instância de consenso (linha 14), o servidor inicia o protocolo de consenso (linha 15). Ao término do protocolo de consenso,o resultado é repassado aos clientes através do `Multisend` da mensagem $\langle \text{RESULT}; idC; agreedVector; sign_i \rangle$ (linha 21). Os servidores constroem um vetor baseado nas propostas recebidas dos clientes. Este vetor é certificado através de um conjunto de mensagens digitalmente assinadas pelos clientes. O protocolo de consenso usado pelos servidores é uma variação do problema clássico de consenso chamado Consenso com Valor Inicial Certificado. Esta variação do consenso e um algoritmo para resolvê-lo na presença de faltas Bizantinas é apresentada em [24]. Este problema garante que um valor decidido é certificado.

Para satisfazer as necessidades dos servidores e garantir a propriedade de validade de vetor necessária, o vetor deve ser certificado. Um vetor certificado é descrito na definição 4.1.1. Esta definição assegura que existem $2f + 1$ entradas no vetor, cada uma proposta por um cliente ou $\perp$ caso o cliente correspondente não tenha proposto.

Algorithm 8 Processo cliente c_i do SGC

Constants:
1: f_s : maximum number of faulty servers
Init:
2: $proposal_i \leftarrow 0$
3: $sign_i \leftarrow 0$
4: $resp_i \leftarrow \perp$
5: $v \leftarrow \langle \perp; \perp; \ldots; \perp \rangle$
6: $received\_msgs_i^{idC} \leftarrow \emptyset$
7: $validMessage \leftarrow false$
8: **on *Rdelivery*** $\langle$SYNC; $idC\rangle$ **from** $Iniciador$
9: $proposal_i \leftarrow$ CALCULATEPROPOSAL(idC)
10: $sign_i \leftarrow$ SIGN$(\langle idC; proposal\rangle)$
11: $req_i \leftarrow \langle$PROPOSE; $idC; proposal; sign\rangle$
12: $MultiSend\ req_i\ to\ SGC$
13: **on receive** $\langle$DECIDE; $idC; v; sign_j\rangle$ **from** s_j
14: $validMessage \leftarrow$ VERIFIEDSIGNATURE$(sign_j, s_j)$
15: **if** $validMessage$ **then**
16: $received\_msgs_i^{idC} \leftarrow received\_msgs_i^{idC} \cup \{v\}$
17: **if** $(\#_v received\_msgs_i^{idC} = f_s + 1)$ **then**
18: $result_i \leftarrow$ RESULT(v)
19: $AgreementFinished(idC; result_i)$
20: **end if**
21: **end if**

Definição 4.1.1. *Um processo servidor s_i considera um vetor $vector_i^{idC}$ certificado em respeito a instância de consenso idC, se e somente se:*

1. $\#_\perp vector_i^{idC} \leq f_c$
2. $\forall j \in [1, n_c], \exists msg_j \in cert_i^{idC} : vector_i^{idC}[j] \neq \perp$
   $\Rightarrow msg_j.proposal = vector_i^{idC}[j]$
   $\wedge$VERIFIEDSIGNATURE$(msg_i.sign, c_j)$
   $\wedge msg_j.idC = idC$

4.2 Propriedades do SGC

Os algoritmos do SGC, mesmo sendo genéricos, fornecem uma série de propriedades e características úteis para a prova de corretude de protocolos e problemas construídos sobre ele. Garantias de liveness, acordo e propriedades do filtro de consenso são feitas. No que segue, nós iremos listar alguns lemas e propriedades juntamente com **sketches** de suas provas.

4.2.1 Lemas gerais

Lema 4.2.1. *Se um servidor correto s_i propõe um vetor na linha 15, então este vetor é válido.*

Algorithm 9 Processo servidor s_i do SGC

Constants:
1: f_c : maximum number of faulty clients
2: n_c : number of clients
Init:
3: $cert_i^{idC} \leftarrow \emptyset$
4: $vector_i^{idC} \leftarrow \langle \perp; \perp; \ldots; \perp \rangle$
5: $agreedVector_i^{idC} \leftarrow \langle \perp; \perp; \ldots; \perp \rangle$
6: $resp\_sign_i \leftarrow 0$
7: $validMessage \leftarrow false$
8: **on receive** $\langle$PROPOSE$; idC; proposal_j; sign_j\rangle$ **from** c_j
9: $validMessage \leftarrow$ VERIFIEDSIGNATURE$(sign_j, c_j)$
10: **if** $validMessage \wedge vector_i^{idC}[j] = \perp$ **then**
11: $vector_i^{idC}[j] \leftarrow proposal_j$
12: $prop_i^{idC} \leftarrow \{\langle$PROPOSE$; idC; proposal_j; sign_j\rangle\}$
13: $cert_i^{idC} \leftarrow cert_i^{idC} \cup prop_i^{idC}$
14: **if** $(\#_\perp vector_i^{idC} = f_c)$ **then**
15: PROPOSE$(idC; vector_i^{idC}; cert_i^{idC})$
16: **end if**
17: **end if**
18: **on decide** $\langle idC; agreedVector_i\rangle$
19: $resp\_sign_i \leftarrow$ SIGN$(\langle idC; agreedVector_i\rangle)$
20: $msg_i \leftarrow \langle$DECIDE$; idC; agreedVector_i; resp_sign_i\rangle$
21: **MultiSend** msg_i **to** *clients*

Demonstração. Sabemos pela definição 4.1.1 que um vetor é válido, se satisfaz as duas condições abaixo:

1. $\#_\perp vector_i^{id} \leq f_c$
2. $\forall j \in [1, n_c], \exists msg_j \in cert_i^{idC} : vector_i^{idC}[j] \neq \perp$
   $\Rightarrow msg_j.proposal = vector_i^{idC}[j]$
   $\wedge$ VERIFIEDSIGNATURE$(msg_i.sign, c_j)$
   $\wedge\, msg_j.idC = idC$

A primeira condição da definição de vetor válido é garantida, pois para o servidor correto s_i executar o protocolo de consenso na linha 15, implica que o teste da linha 14 foi satisfeito, garantindo então que a condição 1 acima é verdadeira ($\#_\perp vector_i^{id} \leq f_c$). A segunda condição da definição 4.1.1 é também garantida na verificação do teste da linha 10, pois:

- A variável $validMessage$ assume valor $true$ na linha proposta 9, o que implica na assinatura da mensagem pelo cliente c_j e na validade desta assinatura.
- Também como conseqüência da verificação deste teste, na linha 11, a proposta de c_j é atribuída a posição j de $vector_i^{idC}$.

- E a mensagem msg_j ($\langle$PROPOSE; id; $proposal_j$; $sign_j\rangle$), enviada com a proposta de c_j, é incluída no certificado $cert_i^{idC}$ do vetor $vector_i^{idC}$ (adicionada ao certificado na linha 13).

Como as duas condições da definição 4.1.1 são verificadas a partir do algoritmo 9, então o vetor proposto na linha 15 por um servidor correto é certificado (válido). □

Lema 4.2.2. *Todo processo servidor correto decidirá um mesmo vetor decisão.*

Demonstração. A demonstração deste lema é obvia devido ao uso pelos servidores do SGC de um protocolo de consenso. Um processo servidor correto decide um vetor, enviando o mesmo em *MultiSend* aos clientes na linha 21. O vetor enviado é o atribuído à variável $agreedVector$ na linha 15, como resultado do protocolo de consenso usado pelos servidores. Devido a propriedade de acordo deste protocolo de consenso executado, todo servidor correto atribuirá o mesmo valor a variável $agreedVector$. Logo, concluí-se que todo servidor correto decidirá um mesmo valor de vetor. □

4.2.2 Propriedades de Terminação

Lema 4.2.3. *Se um servidor correto s_i inicia um Consenso de Valor Inicial Certificado na linha 15 do algoritmo 9, então este consenso irá terminar.*

Demonstração. Assumimos como premissa que o algoritmo de consenso usado no SGC satisfaz as propriedades de acordo, terminação e validade. Sendo s_i correto e o vetor proposto por s_i na linha 15 certificado (lema 4.2.1), então temos um *Consenso de Valor Inicial Certificado* e o algoritmo solução termina se ao menos uma das propostas iniciais é certificada. Logo, existe uma proposta inicial certificada e o algoritmo de consenso terminará. □

Lema 4.2.4. *Se um servidor correto recebe $n_c - f_c$ propostas válidas de clientes distintos com mesmo idC de instância de consenso, então $\#_{\perp} vector_i^{idC} = f_c$.*

Demonstração. Cada proposta válida recebida de um cliente c_j ($msg_j.proposal$) é atribuída a $vector_i^{idC}[j]$ na linha 11. Após a recepção de $n_c - f_c$ mensagens distintas e válidas dos processos clientes, $n_c - f_c$ posições diferentes de $vector_i^{id}$ são preenchidas.

Sabendo que o tamanho do vetor é n_c, então o número de posições vazias em $vector_i^{idC}$ (preenchidas com $\perp$ é igual a $n_c - (n_c - f_c) = f_c$. □

Lema 4.2.5. *Todo servidor correto que recebe $n_c - f_c$ propostas válidas de clientes, acabará decidindo um vetor e o enviará a todos os clientes.*

Demonstração. Suponha que exista um servidor correto s_i que tenha recebido $n_c - f_c$ propostas válidas de clientes distintos e não decida um vetor. Logo este servidor nunca executará o *MultiSend()* na linha 21, responsável pelo envio da mensagem de resposta aos clientes. Isto implica que uma das condições abaixo é verdadeira:

1. Algoritmo de consenso utilizado na linha 15 não termina.

2. O teste encontrado na linha 14 do algoritmo ($\#_{\perp} vector_i^{idC} = f_c$) nunca será satisfeito.

A afirmativa 1 é falsa devido ao lema 4.2.3. Em relação à condição 2 acima, por hipótese, o servidor recebeu $n_c - f_c$ propostas válidas de clientes distintos. Logo, devido ao lema 4.2.4, a condição ($\#_{\perp} vector_i^{idC} = f_c$) é verdadeira e o teste da linha 14 é satisfeito. Portanto, esta condição 2 é também falsa.

Como ambas as condições 1 e 2 são falsas, então todo servidor correto recebe $n_c - f_c$ propostas válidas de clientes distintos acabará executando *MultiSend()* na linha 21. Logo todo servidor correto s_i recebe $n_c - f_c$ propostas válidas de clientes distintos e acaba decidindo um vetor e enviando o mesmo a todos os clientes. □

Teorema 4.2.6. *Se um iniciador começa uma instância de acordo entre n_c clientes, então todos os clientes corretos acabarão decidindo um valor.*

Demonstração. Um iniciador começa um acordo através de um *multicast* confiável da mensagem $\langle \text{SYNC}; idC \rangle$ para o conjunto de clientes (linha 3 do algoritmo 7). Devido à propriedade de *agreement* e de validade do *Rmulticast* se um cliente correto recebe a mensagem $\langle \text{SYNC}, idC \rangle$, então todos processos clientes corretos também receberão esta mensagem. Como entre os n_c clientes no máximo f_c são faltosos então, através da linha 12 do algoritmo 8, no mínimo $n_c - f_c$ mensagens $\langle \text{PROPOSE}; idC; proposal_i; sign_i \rangle$ são enviadas por clientes corretos.

Todo servidor recebe no mínimo $n_c - f_c$ propostas (oriundas dos clientes corretos) devido a característica da primitiva *MultiSend* e dos canais ponto a ponto confiáveis. Segundo o lema 4.2.5, todo servidor correto decide um valor e o envia aos clientes. Como existem $n_s - f_s > f_s + 1$ servidores corretos, todo cliente receberá no mínimo $f_s + 1$ respostas de servidores corretos. Pelo lema 4.2.2, todos os valores decisão recebidos de servidores corretos são idênticos. Assim sendo, o teste da linha 14 será satisfeito e o método AgreementFinished, que caracteriza a decisão do cliente, acabará sendo executado. □

4.2.3 Propriedades de Acordo

Teorema 4.2.7. *Todo processo cliente correto decidirá um mesmo valor.*

Demonstração. O cliente aguardará a chegada de $f_s + 1$ vetores de decisão v idênticos na linha 17 (algoritmo 8 dos clientes). Sabe-se que existem no máximo f_s servidores maliciosos então, entre quaisquer $f_s + 1$ vetores recebidos, pelo menos um foi enviado por um servidor correto. Devido ao lema 4.2.2, todo servidor correto decide o mesmo vetor v. Portanto, qualquer subconjunto de tamanho f_s+1 constituída de respostas idênticas provenientes de servidores possui o mesmo vetor v. Com isto, todo cliente correto possuirá o mesmo v na linha 18. Como a função $Result$ do filtro de consenso é determinista e opera sobre o mesmo domínio v nas execuções do algoritmo 8 pelos diversos clientes, então o mesmo valor será calculado por todos os clientes corretos. Com isto, a variável $result$, que possui o valor decidido por um cliente, possuirá o mesmo valor em todos os clientes corretos. □

4.2.4 Propriedades do Filtro de Consenso

Lema 4.2.8. *Todo componente não-nulo do vetor que é domínio na função $Result$ foi proposta por um cliente.*

Demonstração. Suponha que a proposição seja falsa. Neste caso, existe um componente não nulo que não foi proposto por um cliente no vetor que é domínio da função $Result$, filtro de consenso da linha 18 do algoritmo 8. Para aplicar esta função, um cliente c_i deve receber $f_s + 1$ mensagens DECIDE de servidores com o mesmo vetor v. Estas mensagens são enviadas pelos servidores corretos na linha 21 do algoritmo 9 (algoritmo dos servidores). O vetor enviado por servidores corretos é aquele calculado na linha 15 (algoritmo 9), que é resultado do protocolo de consenso executado entre os servidores. Sabe-se que este protocolo de consenso irá ter como valor de decisão um vetor certificado (*Consenso de Valor Inicial Certificado*).

Se o vetor v é certificado, então este possui uma mensagem PROPOSE ($\langle$PROPOSE$; idC; proposal_j; sign_j\rangle$) assinada por c_j, em toda entrada j, com valor diferente de $\perp$ em v. Assinaturas não podem ser forjadas, portanto toda entrada deste vetor foi proposta por um cliente. Então a afirmativa inicial, de que uma componente de v não foi proposto por um cliente, é um absurdo, logo se conclui que todo componente diferente do default ($\perp$) no vetor em que se aplica a função $Result$ (filtro de consenso) foi proposta de um cliente. □

Teorema 4.2.9. *Todo vetor decidido por um servidor correto possui no mínimo $f_c + 1$ elementos propostos por clientes corretos.*

Demonstração. Todo servidor correto irá decidir na linha 21 o vetor acordado entre servidores na linha 15. Sabemos que o vetor resultante da execução do

protocolo de consenso da linha 15 é certificado (Consenso com Valor Inicial Certificado), logo $\#_{\perp} vector_i idC \leq f_c$ é verdadeiro. Isto garante que no mínimo $n_c - f_c$ valores diferentes do default ($\neq \perp$) estão presentes no vetor. Sabendo que $n_c \geq 3.f_c + 1$ então $n_c - f_c \geq 2.f_c + 1$. Dentre os n_c clientes no máximo f_c são maliciosos, e os valores destes podem estar contidos entre os valores escolhidos pelo servidor para montar o seu vetor. Assumindo o pior caso onde todos os f_c valores enviados por clientes maliciosos estão contidos no vetor, mesmo assim sobram como valores corretos no mínimo $(2f_c + 1) - f_c = f_c{+}1$. Assim, se as propostas contidas no vetor são oriundas dos clientes, no mínimo $f_c + 1$ destas entradas são provenientes de clientes corretos.

Do exposto acima e do lema 4.2.8, pode-se concluir que todo vetor decidido por um servidor correto possui no mínimo $f_c{+}1$ elementos propostos por clientes corretos. □

Corolário 4.2.10. *Vetores usados como domínio da função $Result$ (filtro de consenso, linha 18 do algoritmo 8) possuem $f_c + 1$ elementos provenientes de clientes corretos.*

Demonstração. A função $Result$ de clientes corretos possui como domínio o vetor acordado entre os servidores. Isto é verdade, pois o algoritmo do cliente espera $f_c + 1$ mensagens com mesmo vetor v. Qualquer $f_s + 1$ mensagens sempre terá pelo menos um servidor correto decidiu este valor. Como são $f_s + 1$ os servidores corretos e, segundo o lema 4.2.4, servidores corretos terminam decidindo um mesmo vetor v e enviam este para os clientes. Devido ao teorema 4.2.9, v possui no mínimo $f_c + 1$ elementos propostos por clientes corretos. □

A prova deste corolário termina as provas de que o problema do SGC satisfaz as propriedades de consenso. Foi assumida neste trabalho, a propriedade de validade do consenso de vetores que Dodou e Schiper introduziram em [24, 46] e que é suficientemente forte na presença de faltas bizantinas, permitindo a redução de outros problemas de acordo para o consenso. Esta propriedade de validade tem a sua garantia afirmada pelas provas referentes ao filtro de consenso apresentadas nesta seção 4.2.4.

4.3 Aplicações

Nesta seção, alguns exemplos de uso do SGC para resolução dos problemas de acordo apresentados na seção 2.2 são apresentadas. A princípio o modelo de faltas adotado é bizantino, caso não seja será explicitado. Este é o caso de alguns problemas de acordo que apresentam impossibilidade de resolução em um ambiente bizantino, como *atomic commit*.

4.3.1 Consenso de Vetor

No problema de consenso de vetor (apresentado na seção 2.2.3), um conjunto de n processos acordam um vetor de tal forma que cada entrada deste vetor corresponde a um valor proposto por um processo. Os processos participantes do consenso são numerados de $[1..n]$, a entrada i do vetor acordado é referente ao processo p_i. O que caracteriza a corretude de um vetor resultante deste protocolo é a existência de pelo menos $f + 1$ propostas de processos corretos dentro do vetor decidido.

Para o problema do consenso de vetor, o iniciador foi assumido como um dos clientes que dispara a instancia do consenso. Os clientes são os processos que irão propor o valor inicial do consenso. A função Result (filtro de consenso) correspondente é mostrada no algoritmo 10. Os algoritmos de clientes e iniciador permanecem inalterados em relação a outros aspectos neste problema.

Este problema é resolvido através de uma degeneração da função $Result$ do filtro de consenso, onde $Result$ é a função identidade. Isto é, sendo A o conjunto de todos os vetores de tamanho n_c, $Result$ é a função de $f : A \rightarrow A, f(x) = x$. Intuitivamente é possível perceber que a função identidade resolve o problema de vetor de consenso, pois o vetor recebido na função $Result$ já satisfaz as propriedades de validade de vetor requeridas. O restante das propriedades são garantidas pela estrutura geral do SGC.

Algorithm 10 Função $Result$ do Problema Vetor de Consenso

```
1: function RESULT(vector)
2:     return vector
3: end function
```

Corretude

A infraestrutura fornecida pelo SGC aliada ao Filtro de Consenso definido no algoritmo 10 obedecem as propriedades definidas para o problema de *vector consensus*. Abaixo estão as provas de corretude do algoritmo, baseadas nos teoremos gerais providos pelo SGC.

Terminação: a propriedade de terminação do problema de consenso de vetor segue do teorema 4.2.6, ou seja, se uma instância de consenso é iniciada então todo cliente correto acabará decidindo um valor.

Acordo: a propriedade de acordo segue do teorema 4.2.7 que garante que todo cliente correto decide um mesmo vetor.

Validade de vetor: a propriedade de validade de vetor segue do corolário 4.2.10. Segundo este corolário, pelo menos $f_c + 1$ componentes do vetor domínio da função $Result$ foram propostos por processos corretos. Como este vetor será o valor de decisão de clientes corretos do SGC, a propriedade

de validade de vetor é satisfeita.

4.3.2 Strong Consensus

O problema de *strong consensus* é uma variação do problema de consenso (descrito na subseção 2.2.2 do capítulo 2) no qual um conjunto de n processos acordam um único valor. A particularidade que caracteriza este problema específico de consenso, é o fato de que caso todos os processos corretos proponham o mesmo valor (escalar) v, então todo processo correto decide v.

Para o problema do Strong Consensus (SC), o iniciador foi assumido como um dos clientes que dispara a instancia do consenso. Os clientes são os processos que fazem as propostas iniciais ao Strong Consensus. A função Result (filtro de consenso) correspondente é mostrada no algoritmo 11. Os algoritmos de clientes e iniciadores também permanecem inalterados em relação a outros aspectos neste problema.

Algorithm 11 Função $Result$ do Problema Strong Consensus

```
1: function RESULT(vector)
2:     return MAJORITY(vector)
3: end function
```

Corretude

Abaixo segue a prova de corretude do algoritmo de *strong consensus* apresentado anteriormente.

Terminação: a terminação do algoritmo segue do teorema 4.2.6: se um acordo é iniciado então todo cliente correto acabará decidindo um valor.

Acordo: a propriedade de acordo segue do teorema 4.2.7, o qual estipula que todo cliente correto decidirá um mesmo valor.

Validade: a propriedade de validade na sua verificação exige a separação em dois casos:

1. O valor de decisão foi proposto por um cliente, e

2. Todos os processos corretos propõem um mesmo valor v, e o valor de decisão é v.

No caso 1, a propriedade de validade do problema de *Strong Consensus* especifica que o valor decidido foi proposto por algum cliente. No caso do *Strong consensus* implementado a partir do SGC, isto é verificado, pois segundo o lema 4.2.8, toda entrada do vetor recebido na função $Result$ do filtro de consenso foi proposta por um cliente. Como a função $Majority$ irá retornar o valor mais freqüente dos propostos, o valor de decisão do cliente será um valor proposto por um cliente.

Para provar o segundo caso usa-se contradição. Suponha que a proposição seja falsa, i.e., todos os clientes corretos propuseram um mesmo valor v e um valor v' foi decidido pelos mesmos clientes através do SGC. Para um cliente decidir v' na linha 19 do algoritmo 8 é necessário que este cliente tenha recebido um vetor de decisão dos servidores contendo uma maioria de valores v'. Sabe-se, do corolário 4.2.10, que $f_c + 1$ entradas neste vetor foram propostas por clientes corretos. Como todo cliente correto propôs o mesmo valor v (condição 2). No vetor, f_c podem ter origem em processos maliciosos, mas o restante deve vir de processos corretos, portanto: $\#_v vector_i id \geq f_c + 1$. Então temos uma contradição e a suposição deve ser negada (valor de decisão assumido como v'). A conclusão que se chega é que se todos os clientes corretos propuserem um mesmo valor v e então o valor de decisão obtido a partir da função $Majority$ (algoritmo 11, linha 2), será v.

4.3.3 Reliable Broadcast

Reliable Broadcast (RB) é um problema de comunicação de grupo cujo objetivo é enviar uma mensagem a todos os processos pertencentes a um dado grupo. A descrição do seu comportamento foi feita na subseção 2.2.4 do capítulo 2.

No que segue, assume-se que os identificadores das mensagens são formados pela concatenação de dois componentes $ID = j|sn$, onde j é o identificador do emissor e sn é o número serial desta mensagem de acordo com o emissor. Isto é feito em [27] e não acarreta perda de generalidade. O algoritmo 12 descreve a implementação do RB utilizando o SGC. E algoritmo 13 descreve o filtro de consenso utilizado.

Algorithm 12 Cliente/Iniciador c_i do *Reliable Broadcast*

Init:
1: $proposal_i^{idC} \leftarrow \perp$
2: **procedure** $R\_Broadcast(msg_i)$
3: $MultiSend\ \langle INITIAL, msg_i \rangle\ to\ clients$
4: **end procedure**
5: **on receive** $\langle INITIAL, msg_j \rangle$ **from** c_j
6: $idC \leftarrow msg_j.id$
7: $proposal_i^{idC} \leftarrow message$
8: REQUIRELOCALSTART(idC)
9: **procedure** CALCULATEPROPOSAL(idC)
10: **return** $proposal_i^{idC}$
11: **end procedure**
12: **procedure** AGREEMENTFINISHED($idC, result$)
13: $RB\_deliver(\langle idC, result_i \rangle)$
14: **end procedure**

No problema de *reliable broadcast*, o conjunto de clientes e inicia-

dores coincide, permitindo que os iniciadores utilizem o segundo método de inicialização de instâncias de acordo, que não exige o uso de um *reliable broadcast*. O iniciador de uma instância de acordo da partida a um acordo enviando a todos os seus pares clientes uma mensagem INITIAL através de um *multisend* (algoritmo 12). Tal mensagem carrega consigo a mensagem da aplicação. Cada mensagem possui um identificador único, como expresso anteriormente, que é utilizado como o identificador da instância de consenso que decidirá qual mensagem é entregue à aplicação por todos os processos. Ou seja, o algoritmo assume que $idC = msg.id$ (ou $idC = j|sn$).

Algorithm 13 Filtro de Consenso do *Reliable Broadcast*

```
1: function RESULT(vector)
2:     return MAJORITY(vector)
3: end function
```

A método $RBroadcast()$ explora a característica do protocolo *multisend* que assegura a entrega da mesma mensagem a todos os processos corretos caso o emissor seja correto. Ao realizar um $RBroadcast()$, o cliente executa um *multisend* da mensagem $\langle$INITIAL; $msg\rangle$, fazendo com que todos os processos clientes corretos recebam a mensagem $\langle$INITIAL; $msg\rangle$, caso o emissor seja correto.

No algoritmo 12, ao receberem uma mensagem INITIAL os clientes c_i's definem suas propostas para o valor da mensagem msg. Executam a função $RequireLocalStart(idC)$(algoritmo 7) na seqüência, dando início a uma instância de consenso no SGC.

Estas propostas dos clientes serão submetidas ao SGC através da função $CalculateProposal(idC)$, chamada pelos clientes no algoritmo8. Os vetores consenso entregues pelos servidores do SGC aos clientes ficam sujeitas a função $Result()$ definida no algoritmo13. Por fim, a chamada pelos clientes da função $AgreementFinished(idC, result_i)$ no algoritmo8, deve determinar a entrega de um mesmo valor de msg nos processos corretos.

Emissores maliciosos não irão conseguir subverter as propriedades do algoritmo. Isto acontece por dois motivos: (1) o serviço de consenso garante o acordo entre os clientes corretos, (2) a definição do problema permite que um broadcast realizado por um processo faltoso não seja entregue ou assuma um valor qualquer. Assim sendo, se um processo faltoso não atinge um número suficiente de mensagens com o *multisend* para o SGC conseguir alcançar o consenso, o protocolo irá apenas ignorar esta mensagem e não entregá-la. Se um número suficiente de mensagens for atingido, então o SGC garante as propriedades de acordo escolhendo uma das mensagens recebidas pelos servidores para ser entregue pelos clientes à aplicação.

Corretude

Abaixo seguem as provas de que o algoritmo obedece as três propriedades que definem o problema de RB.

Validade: suponha que um processo correto p_i realiza R_broadcast de uma mensagem msg identificado por $i|sn$. Isto implica que uma mensagem $\langle INITIAL, i|sn, msg\rangle$ será enviada a todos os processos na linha 3 através de um $Multisend()$. Ao receber esta mensagem todos os $2f_c + 1$ processos corretos irão iniciar a instância de acordo identificada por $idC = i|sn$ e adotam como proposta a mensagem msg recebida de p_i (linhas 7 e 10) . Como existem no máximo f_c clientes faltosos, o vetor no filtro de consenso contém pelo menos $f_c + 1$ propostas oriundas de clientes corretos (teorema 4.2.9), e todos os processos corretos propuseram a mesma mensagem, então existirão pelo menos $f_c + 1$ mensagens idênticas. Como no máximo f_c mensagens são diferentes de msg (oriundas de clientes faltosos) e o filtro de consenso aplica a função $Majority$, a mensagem msg será retornada do FC. Após o processamento do FC, msg é passada ao procedimento AgreementFinished, seguindo o fluxo de execução do SGC, e entregue a aplicação por todos os clientes corretos (teorema 4.2.7).

Integridade: tal propriedade é composta por duas proposições: (1) Seja $i|sn$ um identificador, qualquer processo correto entrega uma única mensagem msg com identificador $i|sn$. (2) Se o processo c_i que enviou $i|sn$ é correto então este processo realizou $Rmulticast$ de msg.

O primeiro item segue diretamente do algoritmo e das propriedades do SGC. Se uma mensagem com identificador $i|sn$ foi entregue na linha 13, então esta mensagem foi o resultado da instância de acordo com identificador id. Devido as propriedades do servidor de consenso, um cliente decide apenas uma vez um valor para uma dada instância do acordo.

A segunda proposição também segue diretamente do algoritmo. Seja msg uma mensagem entregue por algum processo correto c_j enviada pelo processo correto c_i. Para que c_j tenha entregue a mensagem msg na linha 13, então msg foi o resultado do filtro de consenso e consequentemente resultado do acordo. Para que msg seja decida pelo servidor de consenso todos os processos corretos devem enviar sua proposta para os servidores do SGC. Um cliente correto só envia sua proposta ao iniciar uma instancia do consenso, o que ocorre na linha 8 ao receber uma mensagem INITIAL. Os clientes corretos só aceitam uma mensagem INITIAL com identificador $j|sn$ se ela foi enviada por j, como c_j é correto então este processo obrigatoriamente realizou o multisend da linha 3. Isto só pode ocorrer se o processo c_j realizou o R_Broadcast.

Acordo: Esta propriedade e assegurada pelo teorema 4.2.7. Este teorema garante que todo cliente decidirá um mesmo valor. Como no problema

do RB o valor decidido é a mensagem entregue pelos processos a aplicação, todos os clientes corretos entregam o mesmo conjunto de mensagens. No caso de um cliente malicioso tentar realizar um broadcast dois desfechos são possíveis: ou a mensagem não é entregue, ou uma mensagem qualquer é escolhida e entregue por todos os processos corretos. Este comportamento está de acordo com as propriedades estabelecidas.

4.3.4 Atomic Broadcast

Atomic Broadcast (AB) é um problema de comunicação de grupo, no qual, além da garantia de todas as propriedades do RB possui a propriedade de ordenação: todo processo correto deve entregar mensagens na mesma ordem. A descrição do seu comportamento foi feita na seção 2.2.5 do capítulo 2. Assim como no protocolo RB, assumimos que os identificadores das mensagens são formadas pela concatenação de duas partes $ID = j|sn$ onde j é o identificador do emissor e sn um número serial atribuído pelo emissor.

Para o problema do *atomic broadcast* (AB), os clientes são aqueles processos que realizam *broadcast* de mensagens e entregam mensagens para a aplicação. O comportamento dos clientes é mostrado no algoritmo 14 . A função *Result* (filtro de consenso) correspondente é mostrada no algoritmo 15. É importante ressaltar que o serviço de *reliable multicast* utilizado pelo algoritmo 14 pode ser implementado pelo mesmo serviço de consenso que será utilizado para acordar a ordem das mensagens.

Os clientes expõem um método a aplicação, $AB\_Broadcast$, que recebe uma mensagem para ser disseminada e o seu identificador. Esta mensagem deve ser entregue por todos os clientes corretos à aplicação na mesma ordem, através do uso da primitiva $AB\_Deliver$. O algoritmo opera em rodadas assíncronas, sendo que em cada rodada o serviço de consenso é utilizado para acordar um conjunto de mensagens estáveis a serem entregues a aplicação. Uma mensagem é considerada estável por um processo correto caso consiga-se provar que pelo menos um processo correto deseja entregá-la a aplicação. Isto significa que a mensagem foi recebida através do RB e todos os processos a acabarão recebendo, e entregando.

Ao realizar um $AB\_Broadcast$ o cliente utiliza os serviços do RB para disseminar a mensagem para todos os processos corretos. Mensagens entregues pelo RB são capturadas no método $R\_Deliver$ e armazenadas na variável $messagesToDeliver$ (linha 8). Caso uma instância de acordo não esteja ativa, um acordo é iniciado pelo cliente (linha 9). A proposta de um cliente ao SGC conseiste em um conjunto contendo o *hash* das mensagens recebidas através do RB que ainda não foram entregues a aplicação.

Assim como no RB, os servidores irão acordar um vetor com as propostas dos diversos clientes que o receberão no filtro de consenso do algo-

Algorithm 14 Cliente/Iniciador c_i do *Atomic Broadcast*

Init:
1: $msgsToDeliver \leftarrow \emptyset$
2: $current\_id \leftarrow 0$
3: $agreementStartRequired \leftarrow true$
4: **procedure** AB_Send(ID, msg)
5: $R\_Broadcast(ID, msg)$
6: **end procedure**
7: **procedure** $RB\_Deliver$(ID, msg)
8: $msgsToDeliver \leftarrow msgsToDeliver \cup \{\langle ID, msg\rangle\}$
9: **if** $agreementStartRequired$ **then**
10: REQUIRELOCALSTART($current\_id$)
11: $agreementStartRequired \leftarrow false$
12: **end if**
13: **end procedure**
14: **procedure** CALCULATEPROPOSAL(id)
15: **if** $current\_id < id$ **then**
16: **return** $\emptyset$
17: **end if**
18: $proposal \leftarrow \{\text{HASH}(m) : m \in msgsToDeliver\}$
19: **return** $proposal$
20: **end procedure**
21: **procedure** AGREEMENTFINISHED($id, result$)
22: **wait until** $(id = current\_id) \wedge (\forall msg \in result : \text{HASH}(msg) \in msgsToDeliver)$
23: $msgs \leftarrow \{msg \in msgsToDeliver : \text{HASH}(msg) \in result\}$
24: **for all** $msg \in msgs$ **do**
25: $AB\_deliver(msg)$
26: **end for**
27: $msgsToDeliver \leftarrow messageToDeliver - result$
28: $current\_id \leftarrow current\_id + 1$
29: $agreementStartRequired \leftarrow true$
30: **if** $|msgsToDeliver| > 0$ **then**
31: REQUIRELOCALSTART($current\_id$)
32: $agreementStartRequired \leftarrow false$
33: **end if**
34: **end procedure**

ritmo 15. A função deste filtro de consenso toma como entrada um vetor de tamanho n_c e retorna um conjunto ordenado contendo o *hash* de todas as mensagens que foram propostas por $f_c + 1$ clientes. Assim, apenas os *hashes* de mensagens estáveis permanecem. Por fim, os *hashes* das mensagens propostas pelos clientes são recebidas no procedimento $AgreementFinished$ onde as mensagens são entregues à aplicação na mesma ordem em que os *hashes* se encontram. Caso alguma mensagem que deva ser entregue ainda não tenha sido entregue pelo RB, o cliente aguarda a chegada da mesma, prosseguindo com a entrega de mensagens a aplicação. Caso existam mensagens ainda não entregues após todas as mensagens serem entregues, uma nova instância de

Algorithm 15 Filtro de Consenso para Resolução do *Atomic Broadcast*

```
1: function RESULT(vector)
2:     allMsgs ← ⋃_{i=1}^{n_c} vector[i]
3:     deliverableMsgs ← {msg|#_msg allMsgs ≥ f_c + 1}
4:     return DETERMINISTICORDER(deliverableMsgs)
5: end function
```

acordo é iniciada (linha 31).

Corretude

Abaixo seguem as provas de que o algoritmo satisfaz as propriedades exigidas pelo problema de *atomic broadcast*.

Validade: se p_i for um cliente correto, ao realizar o AB_Broadcast de uma mensagem M o cliente irá realizar um $RB\_Broadcast$ (linha 4). Pela propriedade do $RB\_Broadcast$, todos os clientes corretos acabarão entregando a mensagem M e executando o procedimento RB_Deliver (linha 6). As mensagens recebidas através de $RB\_Deliver$, são armazenadas na variável $messagesToDeliver$, que é a proposta que será enviada ao serviço de consenso.

Existe um tempo t no qual o vetor decidido pelo serviço de consenso possui $f_c + 1$ entradas contendo M. Suponha que não exista t. Chamemos t' o tempo no qual todos os $2f_c + 1$ clientes corretos recebem pelo reliable multicast a mensagem M, i.e., $M \in messagesToDeliver$. Mas pelo teorema 4.2.9, $f_c + 1$ entradas do vetor são provenientes de clientes corretos. Como todos os clientes corretos propõe um conjunto de mensagens contendo M, pelo menos $f_c + 1$ entradas do vetor contém M.

Como M estará em f_c+1 entradas do vetor decido pelo serviço de consenso, M estará no conjunto de mensagens a serem entregues criada no filtro de consenso nas linhas 2,3 e 4 do algoritmo 15 e entregues na linha 23. Assim sendo, concluí-se que se um processo correto realizar um AB_Broadcast de uma mensagem M, ela acabará sendo entregue por algum processo.

Acordo: a propriedade de acordo segue diretamente do teorema 4.2.7. Como todo cliente correto decide o mesmo conjunto de mensagens a serem entregues, se um processo entrega uma mensagem M obrigatoriamente todos os outros clientes corretos entregarão M à aplicação.

Integridade: a propriedade de integridade do algoritmo de RB utilizado pelos processos para disseminar a mensagem garante que mensagens não serão modificadas, e que caso o emissor de msg seja correto ele realizou $AB\_Broadcast$ de msg.

Ordem total: a prova se dá por contradição. Suponha que existe um

cliente correto c_i que entrega a mensagem M e em seguida M', e segundo cliente c_j que entrega a mensagem M' e em seguida M. Se um cliente correto entregou M e em seguida M' na linha 25, então existem dois casos a considerar: M e M' foram entregues na mesma instância de consenso ou não.

Se M e M' foram entregues na mesma instância de consenso, então M está antes de M' em $result$. Esta variável contém o valor de decisão atingido pelos clientes. Segundo o teorema 4.2.7, o valor decidido pelos clientes é igual para todos os clientes corretos. Como c_j é um cliente correto na sua variável $result$ possui M e depois M'. Então, c_j entrega M e depois M'. chegando-se a uma contradição.

Se M e M' foram entregues em instâncias de consenso diferentes, então M foi entregue em uma instância de acordo com identificador menor que M'. Pois clientes corretos entregam mensagens da instância de acordo id após as mensagens da instância $id-1$ terem sido entregues (linha 22). Assim, se M foi entregue na instância id e M' na instância id' com $id < id'$ e devido ao teorema 4.2.7 que garante que o conjunto de mensagens nas duas instâncias de acordo id e id' nos dois clientes são iguais, infere-se que c_j irá entregar M e depois M'. O que é uma contradição.

Como nas duas hipóteses se chegou a uma contradição, concluí-se que tanto c_i quanto c_j entregarão M e depois M'.

4.3.5 Group Membership

No problema de *membership*, um dado conjunto de processos acordam um subconjunto destes que estão operacionais. Um processo pode precisar ser removido do grupo se for faltoso (ou suspeito). Um processo pode precisar ser adicionado ao grupo, se for recuperado ou se foi erroneamente suspeito de ser faltoso. O protocolo de membership garante que estas mudanças são percebidas pelos processos consistentemente.

Assume-se que todo processo é munido de um detector de faltas que encapsula a classificação de processo em faltosos e corretos. As suspeitas de um detector podem ser equivocadas e contraditórias, i.e., dois processos corretos podem discordar quando a condição de um processo. O protocolo progride em visões, como descrito no capítulo 2 seção 2.2.6, cada visão é composta por um conjunto de processos considerados corretos.

Para adaptar o problema ao serviço genérico de consenso, serão definidos dois papéis: coordenadores, clientes. Os processos coordenadores implementarão o serviço de membership e assumirão o papel de clientes e iniciadores no SGC. Os conjunto de processos clientes do problema de *membership* são aqueles que utilizarão os coordenadores e querem manter uma visão dos outros processos clientes.

Por questões de simplicidade, processos poderão apenas remover pro-

Algorithm 16 Algoritmo do cliente cm_i do *membership*

Init:
1: $view_i^0 \leftarrow \{p : p$ é cliente do membership$\}$
2: $votes_i^{id} \leftarrow \emptyset$
3: $current\_id \leftarrow 0$

4: **on receive** $\langle$INSTALL; id; $new\_view_j\rangle$ **from** $coordenador_j$
5: $votes_i^{id} \leftarrow votes_i^{id} \cup new\_view_j$
6: **if** $(\#_{new\_view_j} votes_i^{id} = f_{coordenador} + 1)$ **then**
7: $view_i^{id} \leftarrow view_j$
8: **while** $(view_i^{current\_id+1} \neq \perp) \wedge (cm_i \in view_i^{current\_id+1})$ **do**
9: $current\_id \leftarrow current\_id + 1$
10: **end while**
11: **end if**

Task 1:
12: **loop**
13: **wait until** $timer > \Delta$
14: $faulty \leftarrow suspected \cap view_i^{current\_id}$
15: $sign \leftarrow$ SIGN$(\langle current_id; faulty\rangle)$
16: **MultiSend**$\langle$UPDATE; $current\_id$; $faulty$; $sign\rangle$ **to** *coordenadores*
17: **reset** timer
18: **end loop**

cessos suspeitos. Entretanto é trivial a implementação de um protocolo de inclusão. Para isto, basta que um processo que deseja participar do grupo envie uma mensagem com o pedido. Os coordenadores então realizariam um acordo decidindo a inclusão ou não do processo. Caso ele seja aceito, uma mensagem contendo a visão em que ele foi incluso seria enviada assim como mensagens de instalação de visão.

Em linhas gerais o algoritmo funciona da seguinte forma: os clientes do problema de *membership* possuem um detector de falhas capaz de classificar outros clientes em faltosos ou não-faltosos, expondo ao algoritmo uma lista de processos suspeitos $faulty$. A última visão instalada pelo cliente é a visão indicada pela variável $current\_id$, isto é, $view^i_{current\_id}d$. A instalação de uma nova visão se da através do incremento deste ponteiro ($current\_id$) que ocorre na linha 9.

O cliente, de tempos em tempos, calcula quais processos pertencentes a última visão instalada por ele são suspeitos de serem faltosos, através de uma consulta ao seu detector de faltas (linha 14 do algoritmo 16). Estes processos suspeitos são enviados aos coordenadores (linha 15 do algoritmo 16), que tem a responsabilidade de coletar as suspeitas dos processos clientes (linha 7 do algoritmo 17), consultar o SGC e produzir uma nova visão sem aqueles processos garantidamente suspeitos por pelo menos um processo correto. Esta

Algorithm 17 Algoritmo do coordenador c_i do problema do *group membership*

Init:
1: $view_i^0 \leftarrow \{p : p$ é cliente do *membership* $\}$ ▷ visão dos clientes pelo servidor
2: $changes_i^{id} \leftarrow \emptyset$ ▷ conjunto de mensagens UPDATE
3: $current\_agreement\_id \leftarrow 0$ ▷ identificador do próximo acordo a ser realizado
4: **on receive** $\langle$UPDATE$; id; faulty_j; sign_j\rangle$ **from** *cliente* cm_j
5: $valid \leftarrow$ VERIFIEDSIGNATURE$(sign_j, cm_j)$
6: **if** $valid$ **then**
7: $changes_i^{id} \leftarrow changes_i^{id} \cup \langle$UPDATE$; id; faulty_j; sign_j\rangle$
8: **if** $|changes_i^{id}| = \lceil(2|view_i^{id}| + 1)/3\rceil$ **then**
9: REQUIRELOCALSTART(id) ▷ Inicia o acordo em c_i.
10: **end if**
11: **end if**

12: **procedure** CALCULATEPROPOSAL(id)
13: **return** $changes_i^{id}$
14: **end procedure**

15: **procedure** AGREEMENTFINISHED$(id, result)$
16: **wait until** $current\_agreement\_id = id$
17: $f \leftarrow \lfloor(|view_i^{id}| - 1)/3\rfloor$ ▷ máximo de processos faltosos na visão id
18: $faulty \leftarrow \{msg.faulty : msg \in result \wedge |\{msg' \in result : msg'.faulty = msg.faulty\}| > f\}$
19: $view_i^{id+1} = view_i^{id} - faulty$ ▷ remove processos considerados faltosos por mais de f processos
20: **MultiSend** $\langle$INSTALL$; id + 1; view_i^{id+1}\rangle$ **to** $view_i^{id+1}$
21: $current\_agreement\_id = current\_agreement\_id + 1$
22: **end procedure**

nova visão é enviada aos clientes (linha 20 do algoritmo 17). Os clientes, ao receberem $f + 1$ mensagens de instalação de visão iguais dos coordenadores, assumem como esta visão como correta atribuindo esta visão à variável $view_p^{id}$ (linha 7 do algoritmo 16). Em seguida, os clientes instalam as visões (linha 9) já aceitas até que o cliente seja removido da visão ou a próxima visão ainda não tenha sido recebida.

Corretude

Abaixo seguem as provas de que o algoritmo satisfaz as propriedades exigidas pelo problema de *group membership*.

Singularidade (*Uniqueness*): prova segue por redução ao absurdo. Dado que dois processos corretos p_i e p_j instalam duas visões V_i^x e V_j^x respectivamente, suponha $V_i^x \neq V_j^x$. Para p_i instalar a visão V_i^x, p_i recebeu $f_{coordenadores} + 1$ mensagens de coordenadores distintos requisitando

Algorithm 18 Algoritmo função result para problema de *membership*

```
1: function RESULT(vector)
2:     allMsgs ← ⋃_{i=1}^{n_c} vector[i]
3:     validMsgs ← {msg : msg ∈ allMsgs ∧ VERIFIEDSIGNATURE(msg.sign)}
4:     return validMsgs
5: end function
```

a instalação V_i^x (linha 6 do algoritmo 16). Coordenadores corretos enviam a mensagem requisitando a instalação de uma nova visão ao terminar um acordo do SGC (linha 20 do algoritmo 17). Dado que todos os clientes corretos do SGC decidem o mesmo valor (teorema 4.2.7), concluí-se que todos os coordenadores corretos (clientes do SGC) requisitam a instalação da visão $view_i^x$.

Isto ocorre pois: (1) as operações realizadas para o calculo da próxima visão em `AgreementFinished` são determinístas, (2) todos os coordenadores corretos possuem um mesmo conjunto $view_i^0$ e (3) o SGC garante, através do teorema 4.2.7, que todos os clientes decidem o mesmo valor (logo, domínio da função `AgreementFinished` é idêntico entre todos os processos corretos).

Seguindo o mesmo raciocínio, para o processo p_j: $f_{coordenadores} + 1$ mensagens de coordenadores distintos requisitando a instalação de V_j^x (linha 20) foram recebidas. Sabe-se que no máximo $f_{coordenadores}$ coordenadores são faltosos. Logo, existe um coordenador correto que requisitou a instalação da visão V_j^x. Isto é uma contradição pois coordenadores corretos não requisitam instalação de visões distintas para um mesmo identificador x, assim a hipótese é falsa e concluí-se que $V_i^x = V_j^x$

Validade: Dado um processo p_i correto e uma visão V_i^x instalada por p_i. Para que a visão V_i^x seja instalada, o cliente executa a linha 9 do algoritmo 17. Esta linha só é executada quando há a recepção de $f_{coordenadores}+1$ mensagens requisitando a instalação da visão. Logo, existe um coordenador correto que requisitou a instalação da visão. A proposição da propriedade de validade é composta por duas partes: (1) $p_i \in V_i^x$ e (2) todo processo correto $p_j \in V_j^x$, V_j^x acabará instalando V_j^x. A proposição 1 é verdadeira pois coordenadores corretos realizam *multisend* apenas para os clientes em V_i^x, logo clientes que não pertencem a V_i^x recebem no máximo f requisições de instalação de visão. A validade da proposição (2) advém da existência de um coordenador correto que envia a requisição de instalação da visão V_i^x. Como um coordenador correto decidiu, sabe-se que a instância de acordo $x - 1$ foi iniciada e, segundo o teorema 4.2.6, todos os coordenadores corretos decidem, causando o envio da requisição de instalação da visão V_i^x. Sabe-se que existem pelo menos $2f + 1$ coordenadores corretos que enviam V_i^x, logo todos os

clientes na visão V_i^x recebem pelo menos $f + 1$ requisições de instalação de visão, ocasionando sua instalação.

Integridade: A propriedade de integridade diz que se um processo é removido de uma visão ($p \in V^x - V^{x+1}$), ele foi classificado como faltoso por algum processo correto. A prova segue por contradição: suponha que existe um cliente p tal que $p \in V^x - V^{x+1}$ e nenhum cliente correto suspeita de p. Se a visão V^x foi instalada pelos clientes, então ela foi proposta por pelo menos $f_{coordenadores} + 1$ coordenadores, logo existe um coordenador correto que calculou V^x. Como um coordenador correto calculou V^{x+1}, a instância de acordo x foi iniciada, e pelos teoremas 4.2.9 e 4.2.6, todos os coordenadores corretos acabam chegando a mesma decisão $result$. O filtro de consenso é aplicado sobre $result$ e um conjunto com as suspeitas válidas de todos os processos é criado (linha 3). Este conjunto é filtrado (linha 18 do algoritmo 17), resultando no conjunto $faulty$ que contém apenas os clientes suspeitos de serem faltosos por outros $f_c + 1$ clientes. Estes processos são removidos da visão V^x e a visão V^{x+1} é criada (linha 19). Logo, como $p \in V^x - V^{x+1}$, infere-se que $p \in faulty$. Portanto, $f + 1$ clientes enviaram suspeita de que p é faltoso. Entretanto, existem no máximo f processos faltosos, portanto existe um cliente correto que suspeita de p chegando-se a uma contradição.

Vivacidade: A propriedade de vivacidade requer que uma nova visão seja instalada caso um processo $q \in V^x$ seja suspeito por $\lfloor(|V^x| - 1)/3\rfloor + 1$ processos de V^x. Esta propriedade é trivialmente satisfeita pelo algoritmo, pois novas visões são instaladas continuamente. Caso $\lfloor(|V^x| - 1)/3\rfloor + 1$ suspeitam de q, uma nova visão instalada após um certo intervalo de tempo (linha 13 do algoritmo 16), quando todos os processos corretos enviarem suas suspeitas aos coordenadores (15).

4.3.6 Non Blocking Atomic Commit

Uma transação distribuída engloba vários nós em uma rede de computadores, mantendo as propriedades ACID. Uma das questões mais interessantes neste cenário é garantir a atomicidade, isto é, ou todos os processos realizam com sucesso a operação ou nenhum o realiza, como se a operação nunca existisse. O protocolo responsável por garantir a atomicidade é o serviço de *commit* distribuído.

No algoritmo que segue processos são classificados em dois papéis: coordenadores e participantes. Os coordenadores recebem votos dos participantes e decidem se estes devem executar a operação ou abortá-la. Os participantes são aqueles processos que desejam executar uma operação de forma transacional. Assume-se que existe um número finito e bem conhecido de participantes (n_p) e de coordenadores(n_c). Uma transação será sempre entre todos os participantes, isto é, não é possível definir um subconjunto dos

participantes para realizar uma transação. Trabalhos futuros poderão ser desenvolvidos com o objetivo de relaxar esta restrição.

Os coordenadores estão sujeitos a faltas bizantinas. Entretanto, os participantes podem sofrer apenas faltas de *crash*. Assume-se este modelo de faltas para os participantes devido a impossibilidade de resolução do NBAC em ambientes bizantinos indicada em [23].

Algorithm 19 Algoritmo do participante p_i do Atomic Commit

Init:
1: $votes_i^{id} \leftarrow \emptyset$ ▷ armazena votos dos coordenadores para a operação identificada por id
2: **function** NBAC($id, votes_i$)
3: $MultiSend\ \langle$VOTE; id; $vote_i\rangle$ *to coordenadores*
4: **wait until** $\#_{commit} votes_i^{id} > f_c + 1$ **or** $\#_{abort} votes_i^{id} > f_c + 1$
5: **return** $\#_{commit} votes_i^{id} > f_c + 1$
6: **end function**

7: **on receive** $\langle$RESPONSE; id; $vote_i\rangle$ **from** coordenador c_j
8: **if** $\nexists msg \in votes_i^{id} : msg.sender = c_j$ **then**
9: $votes_i^{id} \leftarrow votes_i^{id} \cup \langle$RESPONSE; id; $vote_i\rangle$
10: **end if**

Assume-se que os processos coordenadores possuem um detector de faltas *?P* responsável por detectar a existência de participantes faltosos, a existência de tal detector é uma condição necessário necessário para resolução do problema de NBAC [29]. Este detector de faltas expõe um valor booleano indicando se algum dos participantes sofreu falta de *crash*, e obece as propriedades de (1) *Anonymous Completeness* e (2) *Anonymous Accuracy*. Ou seja, (1) caso algum participante sofra uma falta de *crash*, existe um instante a partir do qual todo processo correto detecta a falta, e (2) nenhuma falta é detectada até que algum participante realmente sofra uma falta.

No contexto do SGC, os processos coordenadores assumem o papel de clientes e iniciadores no SGC. Os clientes são aqueles que votam na execução ou não da transação. O algoritmo funciona da seguinte forma: os participantes da transação identificada por id votam através do método `NBAC` definido no algoritmo 19. Estes votos são enviados aos coordenadores através do *multisend* da mensagem $\langle$VOTE; id; $vote_i\rangle$. O participante aguarda a chegada de $f_c + 1$ mensagens contendo a mesma decisão dos coordenadores (linha 4), assumindo tal decisão como resultado. Os coordenadores seguem o algoritmo 20. Eles aguardam a chegada de votos de todos os participantes ou o detector de faltas $?P$ indicar que houve alguma falta (fazendo a variável $?P_i = True$). Ao coletar todas os votos ou o detector assumir valor verdadeiro, o coordenador utiliza o serviço de consenso para decidir se a transação deve ser efetivada ou

Algorithm 20 Algoritmo do coordenador c_i do problema de *Atomic Commit*

Init:
1: $op_i^{id} \leftarrow [\perp, ..., \perp]$ ▷ votos dos participantes para operação id
2: $?P_i \leftarrow False$ ▷ saída do detector de faltas de c_i
3: **on receive** $\langle$VOTE; id; $vote_j\rangle$ **from** $participante\ p_j$
4: $op_i^{id}[j] \leftarrow vote_j$
5: **if** $\#_{\perp} op_i^{id} = 0$ **or** $?P_i = T$ **then**
6: REQUIRELOCALSTART(id) ▷ Inicia acordo no processo c_i
7: **end if**

8: **procedure** CALCULATEPROPOSAL(id)
9: **if** $?P_i = T$ **or** $\exists x \in op_i^{id} : x =$ ABORT **then**
10: **return** ABORT ▷ Existe algum processo faltoso, ou voto ABORT foi recebido
11: **end if**
12: **return** COMMIT ▷ Todos os processos votaram COMMIT
13: **end procedure**

14: **procedure** AGREEMENTFINISHED($id, result$)
15: $MultiSend$ $\langle$RESPONSE; id; $result\rangle$ $to\ participantes$
16: **end procedure**

não. O filtro de consenso (algoritmo 21) utiliza a função *majority* para escolher a proposta adotada como resultado do NBAC. Este filtro possui a propriedade de que caso todos os processos corretos tenham proposto o mesmo valor, este será escolhido como $result$ a ser usado como domínio da função `AgreementFinished`. Esta função envia o valor decidido aos participantes.

Algorithm 21 Algoritmo função result para problema de commit distribuído

1: **function** RESULT(vector)
2: **return** MAJORITY($vector$)
3: **end function**

Corretude

A seguir, prova-se que o algoritmo delineado satisfaz as propriedades do NBAC descritas na seção 2.2.7 do capítulo 2.

Acordo: A prova se da por contradição. Suponha que dois participantes, p e p', decidem ABORT e COMMIT, respectivamente. Assim, p recebeu $f+1$ votos dos coordenadores informando que a transação deve ser abortada, enquanto p' recebeu f_c+1 votod de que a transação deve ser efetivada. Assim, existe um coordenador correto que enviou voto de ABORT, enquanto um outro coordenador correto enviou voto de COMMIT. Um coordenador correto, en-

via o voto calculado pelo filtro de consenso, como resultado de uma consulta ao SGC. Entretanto, segundo o teorema4.2.7 dois coordenadores não podem decidir valores diferentes, chegando-se a uma contradição.

Validade-*abort*: Caso um participante p_j vote ABORT, em todos os coordenadores corretos $op_i^{id}[j] = ABORT$ (linha 4 do algoritmo 20). Após todos os votos serem coletados, o SGC é consultado na linha 6 utilizando a proposta calculada pelo procedimento `CalculateProposal`. Devido ao fato de que $op_i^{id}[j] = ABORT$ em todos os coordenadores, o teste na linha 9 é verdadeiro e `CalculateProposal` retorna ABORT. Devido ao filtro de consenso utilizado (o mesmo utilizado para resolução de *Strong Consensus*) e como todos os coordenadores corretos propuseram ABORT, o resultado do filtro de consenso é ABORT. Valor este utilizado como domínio da função `AgreementFinished` que resulta em todos os coordenadores enviando aos participantes a decisão ABORT. Como existem mais de f_c coordenadores, todos os participantes irão receber pelo menos $f_c + 1$ votos abortando a transação.

Validade-*commit*: Caso todos os participantes votem ABORT e nenhuma falha ocorra, todos os coordenadores corretos terão apenas votos COMMIT no vetor op_i^{id} (linha 4 do algoritmo 20) e $?P_i = False$ durante toda a execução do algoritmo. Após todos os votos serem coletados, o SGC é consultado na linha 6 utilizando a proposta calculada pelo procedimento `CalculateProposal`. Como todas as entradas de op_i^{id} são iguais a $COMMIT$ e $?P_i = False$ em todos os coordenadores, o teste na linha 9 falha e `CalculateProposal` retorna COMMIT. Devido ao filtro de consenso utilizado (o mesmo utilizado para resolução de *Strong Consensus*) e como todos os coordenadores corretos propuseram COMMIT, o resultado do filtro de consenso é COMMIT. Valor este utilizado como domínio da função `AgreementFinished` que resulta em todos os coordenadores enviando aos participantes a decisão COMMIT. Como existem mais de f_c coordenadores, todos os participantes irão receber pelo menos $f_c + 1$ votos efetivando a transação.

Vivacidade: Devido ao fato de que ou todos os partipantes são corretos e votam, ou o detector de faltas $?P_i$ será $True$, todo coordenador inicia o acordo na linha 6. Segundo o teorema teorema 4.2.6, se a instancia de acordo foi iniciada uma decisão é atingida. Logo, todos os coordenadores corretos acabarão executando o procedimento `AgreementFinished` com mesmo domínio $result$. Os participantes recebem mais de f_c+1 votos já que existem pelo menos $2f_c + 1$ coordenadores corretos, chegando-se a uma decisão.

4.4 Considerações Finais

Neste capítulo foi introduzido o Serviço Genérico de Consenso tolerante a faltas bizantinas, os algoritmos, provas de corretude e algumas aplica-

ções. O SGC favorece a separação de responsabilidades, através da separação entre processos que executam o protocolo de consenso daqueles que desejam resolver um problema de acordo propriamente dito. Esta separação facilita a implementação de problemas de acordo, já que protocolos de consenso não precisam mais serem utilizados por estes, basta uma requisição ao serviço de consenso para resolvê-lo. Além disto, o serviço de consenso pode ser compartilhado entre diversos sistemas distribuídos distintos. Isto traz a vantagem de que atualizações e otimizações realizadas no SGC se tornam imediatamente disponíveis a todos estes sistemas que o utilizam. Por ultimo, os servidores do serviço de consenso estão fisicamente separados dos clientes permitindo a adotação de modelos de faltas distintos e tecnologias não disponíveis nos clientes. Devido a resiliência dos servidores que compõe o SGC ser definida pelo algoritmo de consenso utilizado, algoritmos (como os apresentados no capítulo 5) com uma maior resiliência podem ser utilizados sem que os clientes sejam afetados, permitindo redução dos custos de replicação, por exemplo.

Uma série de outros problemas de acordo podem ser adaptados para a utilização com o SGC. Foram apresentados seis diferentes problemas e seus respectivos algoritmos, e provas de corretude. Estes algoritmos servem como prova da viabilidade de utilização do SGC para resolução de problemas de acordo.

Capítulo 5

Protocolos de Acordo Baseados em Virtualização

Os servidores do SGC utilizam um modelo de faltas híbrido para aumentar a resiliência do protocolo e diminuir o número de réplicas necessárias para resolver o consenso. Isto é realizado através da adoção de componentes confiáveis disponibilizados por um monitor de máquina virtual. Dois componentes distintos são apresentados, ambos apresendo vantagens e desvantagens em termos de implementação e facilidade de utilização.

5.1 Componentes Confiáveis

O modelo de faltas híbrido como proposto neste trabalho é utilizado em diversos trabalhos na literatura [4, 5, 20]. Assume-se que este componente será simples o suficiente para ser desenvolvido utilizando técnicas e métodos formais que garantam a corretude do mesmo sob quaisquer condições.

O componente poderá ser utilizado para comunicação entre processos que desejam atingir o consenso. Ele será acessível por todos os processos, e fornecerá serviços cujo comportamento é definido por algumas propriedades.

Utilizando-se estes componentes confiáveis para realizar a difusão (multicast) de mensagens entre processos, é possível desenvolver um algoritmo no qual o número de processos necessários para tolerar f processos faltosos seja $2f + 1$. Em sistemas assíncronos sujeitos a faltas bizantinas e canais autenticados são necessários no mínimo $3f + 1$ processos para tolerar f processos faltosos. Assim sendo há um aumento da resiliência do sistema com o mesmo número de agentes faltosos.

5.1.1 Postbox

A Postbox é uma abstração que representa uma área onde mensagens são escritas e lidas. O que torna a postbox interessante é que ela faz com que todas as mensagens sejam lidas na mesma ordem por todos os processos. Apesar de ser uma restrição bastante forte, quando os processos que realizam o acordo estão sobre uma mesma máquina física mas em máquinas virtuais distintas este componente passa a ser trivial. Isto porque, corresponde a uma memória append-only compartilhada entre as máquinas virtuais.

São definidas duas operações sobre este componente:

- $enviar(m)$: uma mensagem m, enviada pelo processo p_i que invocou a operação, será difundida entre os processos.
- $lerMsg()$: lê próxima mensagem encontrada na postbox.

Estas operações garantem algumas propriedades básicas de funcionamento, definidas abaixo:

- Integridade: mensagem m é retornada por $lerMsg()$ no máximo uma vez para cada processo que executar $lerMsg()$.
- Validade: se o processo p_i executar a operação $enviar(m)$, então a mensagem m acabará sendo retornada a um processo p_j após um número finito, mas desconhecido, de invocações a $lerMsg()$.
- Acordo: se um processo recebe a mensagem m de $lerMsg()$ então todo processo acabará recebendo m após a execução da operação $lerMsg()$ um número finito, mas desconhecido, de vezes.
- Ordem: se um processo executar a operação $lerMsg()$ duas vezes e obtiver m e m' respectivamente, qualquer processo irá receber m e m' nesta ordem após um número finito, mas desconhecido, de invocações a operação $lerMsg()$.

5.1.2 Postbox distribuída

A Postbox distribuída relaxa algumas propriedades encontradas na postbox. Assim como a postbox, representa uma área onde mensagens são escritas e lidas. A grande diferença é que a postbox não garante que todas as mensagens sejam lidas na mesma ordem por todos os processos, mas garante que a ordem do emissor será respeitada. Assim, um processo malicioso não pode enviar as mensagens m e m' para um processo p enquanto envia m' e m para um processo p'.

São definidas duas operações sobre este componente:

- $Append(m)$: uma mensagem m, enviada pelo processo p_i que invocou a operação, será difundida entre os processos.
- $ReadFromPostbox(i)$: lê mensagem enviada pelo processo p_i.

Estas operações garantem algumas propriedades basicas de funcionamento, definidas abaixo:

- Integridade: mensagem m é retornada por $lerMsg(i)$ no máximo uma vez para cada processo que executar $lerMsg(i)$.

- Validade: se o processo p_i executar a operação $enviar(m)$, então a mensagem m acabará sendo retornada a um processo p_j após um número finito, mas desconhecido, de invocações a $lerMsg(i)$.
- Acordo: se um processo recebe a mensagem m de $lerMsg(i)$ então todo processo acabará recebendo m após a execução da operação $lerMsg(i)$ um número finito, mas desconhecido, de vezes.
- Ordem: se um processo p_i executa as operações $enviar(m)$ e depois $enviar(m')$, então todo processo que executar a operação $lerMsg(i)$ e tiver obtido m', terá já recebido m em uma execução anterior da operação $lerMsg(i)$.

5.1.3 Algoritmos de Consenso

A seguir dois algoritmos de consenso são desenvolvidos utilizando as postbox apresentadas. Com estes algoritmos o número de processos necessário para realizar o acordo é $2f + 1$ onde f é o número de processos faltosos.

5.1.4 Baseado em Postbox

O algoritmo baseado em *Postbox* é o mais simples pois o componente confiável utilizado fornece propriedades mais fortes em relação ao ordenamento das mensagens. O algoritmo 22 funciona da seguinte forma: ao receber uma proposta, todo processo escreve na *postbox* (linha 3). Cada processo é munido de uma tarefa paralela responsável por monitorar a *postbox*. Toda vez que uma proposta válida referente a um consenso ainda não decidido é lida da postbox, esta proposta é decidida (linha 11).

O algoritmo 22 garante as propriedades do consenso. Isto ocorre devido as características da *postbox*. Todo valor é decidido, através da leitura de propostas anexadas na postbox. Como todo processo lê da postbox o mesmo conjunto de mensagens na mesma ordem, a primeira proposta válida lida da postbox será a mesma para todos os processos, garantindo a propriedade de acordo.

5.1.5 Baseado em Postbox Distribuída

Como a postbox distribuída fornece propriedades mais relaxadas do que a postbox, o algoritmo será mais complexo. O algoritmo prossegue em rodadas assíncronas sendo que em cada uma há um processo coordenador. Na rodada r o processo $r \ mod \ n$ será o líder. Logo, o paradigma de coordenador rotativo é utilizado. O líder da rodada propõe um valor para ser adotado como valor de decisão por todos os processos corretos. Após a proposta ser realizada, os processos trocam mensagens entre si, a fim de determinar se podem realmente decidir o valor garantindo as propriedades de acordo.

Algorithm 22 Algoritmo de consenso utilizando a *postbox*

Variables:
1: $decided_i^{id} \leftarrow false$
Algorithm for process p_i:
2: **procedure** PROPOSE($id, proposal, certificate$)
3: APPEND($\langle$PROPOSAL, $id, proposal, certificate\rangle$)
4: **end procedure**
Task 1:
5: **loop**
6: $msg \leftarrow$ READ
7: **if** $msg = \langle$PROPOSAL, $id', proposal', certificate'\rangle$ **then**
8: $valid_i =$ CHECKCERT($proposal', certificate'$)
9: **if not** $decided_i^{id} \wedge valid_i$ **then**
10: $decided_i^{id} \leftarrow true$
11: DECIDE($proposal'$)
12: **end if**
13: **end if**
14: **end loop**

O algoritmo é divido em duas partes. A primeira parte é um módulo de detecção de processos bizantinos. Este módulo monitora as mensagens enviadas por outros processos com o objetivo de detectar comportamentos bizantinos. Este comportamento inclui o envio de mensagens inválidas, em ordem incorreta ou mensagens não condizentes com o histórico de mensagens do processo que a enviou, além de outros comportamentos detectáveis. Não deve-se confundir esta parte do algoritmo com detectores de falhas, este detector não é um componente distribuído como os apresentados em [22, 24, 31, 43, 46]. Este processo toma decisões locais baseadas nas mensagens enviadas para o processo que o está executando. A finalidade é apenas separar este aspecto do algoritmo de forma a simplificar a especificação do algoritmo.

O algoritmo funciona da seguinte forma: no início de cada rodada o coordenador da rodada c envia uma mensagem $\langle$PROPOSAL$; r_i; v; c; init_cert; sign\rangle$ com o número da rodada r_i, o valor proposto v, um certificado $init\_cert$ e a assinatura da mensagem. Na primeira rodada $init\_cert = \emptyset$. Ao receber a proposta cada processo p envia uma mensagem $\langle$PREPARE$; r_i; proposal; sign\rangle$ onde r_i é o número da rodada, $proposal$ é a mensagem PROPOSAL recebida do primário e sign a assinatura da mensagem. Se um processo recebe $f + 1$ mensagens PREPARE de uma mesma rodada com um mesmo valor v, este valor v é decidido e o processo termina, deixando de participar do restante do protocolo. Este é o caso normal do algoritmo, quando nenhum processo faltoso está presente. A figura 5.1 ilusta a troca de mensagens realizadas.

A etapa de prepare é necessária para satisfazer a propriedade de acordo. É garantido pelo componente confiável que todos os processos receberão a

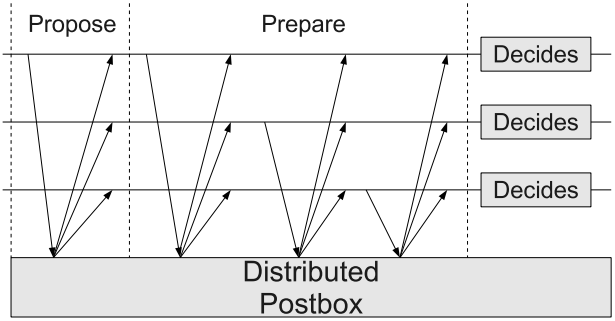


Figura 5.1: Execução boa do algoritmo de consenso baseado em *postbox* distribuída.

mensagem e esta será igual para todos, mas líderes faltosos podem não propor um valor. Assim, é necessário um mecanismo para detectar líderes faltosos e avançar para a próxima rodada. Mas não é possível diferenciar um líder faltoso de um lento, devido a condição FLP. Cada processo decide se o líder é ou não correto utilizando temporizadores locais, logo alguns processos podem decidir que o líder é faltoso enquanto outros não o fazem.

Assim sendo, um processo pode ter tomado uma decisão enquanto outros iniciaram uma nova visão. O protocolo de troca de visão deve levar em consideração esta situação e fazer com que a propriedade de acordo entre rodadas seja garantido. Para isto a seguinte proposição deve ser verdadeira: se um processo pode decidir v na rodada r e um processo correto decidide v' em uma rodada $r' > r$, então $v' = v$.

Quando um processo suspeita que o líder da rodada é faltoso e não irá escrever sua proposta no componente confiável, ou não conseguiu juntar f+1 mensagens PREPARE em tempo hábil, ele envia uma mensagem $\langle$FREEZE; $r_i; v; r_p; init\_cert\rangle$ sendo r_i o número da rodada atual. Para preenchimento dos outros valores existem 2 casos. No primeiro caso o processo enviou uma mensagem PREPARE nesta rodada, enquanto no segundo caso não o fez. No primeiro caso v é o valor contido na mensagem de PREPARE enviada, r_p é a rodada atual e $init\_cert = \perp$. No segundo caso, v é o valor certificado por $init\_cert$ e r_p indica em que rodada este valor foi proposto. Se nenhum valor foi proposto ainda, o que pode acontecer caso o primeiro líder seja faltoso, então $v = \perp$ e $r_p = 0$.

O mecanismo de troca de visão consiste na construção de um $init\_cert$ válido. O $init\_cert$ consiste de $f+1$ mensagens FREEZE coletadas na rodada $r_i - 1$ sem o $init\_cert$ e $f + 1$ hashes assinados de cada mensagem FREEZE.

Algorithm 23 Algoritmo de consenso baseado em Postbox Distribuída

Algorithm for p_i**:**

1: $r_i^{id} \leftarrow 0$ ▷ Rodada atual do processo p_i
2: $init\_cert_i^{id} \leftarrow \langle \emptyset, \perp, \ldots, \perp \rangle$ ▷ Certificado ínicio de rodada
3: **procedure** CONSENSUS($id, proposal, certificate$)
4: **loop**
5: **if** $round_i^{id}$ **mod** $n = i$ **then**
6: WRITEPOSTBOX($\langle$PROPOSAL$, r_i, v_i^{id}, init_cert_i^{id}[r_i^{id}]\rangle$)
7: **end if**
8: **wait until** $proposals_i^{id}[r_i^{id}] \neq \perp$ **or** timed_out
9: $proposal\_received \leftarrow proposals_i^{id}[r_i^{id}] \neq \perp$
10: **if** $proposal\_received$ **then**
11: WRITEPOSTBOX($\langle$PREPARE$, r_i, proposal\rangle$)
12: **end if**
13: **wait until** timed_out ▷ Aguarda tomada decisão
14: **if** $proposal\_received$ **then**
15: $freezeCert \leftarrow proposal$
16: **else**
17: $freezeCert \leftarrow$ last valid proposal received
18: **end if**
19: WRITEPOSTBOX($\langle$FREEZE$, r_i, freezeCert\rangle$)
20: **wait until** $init\_cert_i^{id}[r_i^{id}] \neq \perp$ ▷ Aguarda $init\_cert$
21: $r_i^{id} \leftarrow r_i^{id} + 1$
22: **end loop**
23: **on receive** $\langle$PROPOSAL$, r_j, v_j, init_cert_j^{id}\rangle$ **from** p_j
24: $proposals_i^{id}[r_j] \leftarrow v_j$
25: UPDATEINITCERT(id, r_j)
26: **on receive** $\langle$PREPARE$, r_j, proposal\rangle$ **from** p_j
27: $prepares_i^{id}[r_j] \leftarrow prepares_i^{id}[r_j] \cup \langle$PREPARE$, r_j, proposal\rangle$
28: **if** $\#_{proposal} prepares_i^{id}[r_j] = f + 1$ **then**
29: DECIDE($proposal.v$) ▷ Acordo foi atingido.
30: **end if**
31: UPDATEINITCERT(id, r_j)
32: **on receive** $\langle$FREEZE$, r_j, proposal\rangle$ **from** p_j
33: $freezes_i^{id}[r_j] \leftarrow freezes_i^{id}[r_j] \cup \langle$FREEZE$, r_j, proposal\rangle$
34: UPDATEINITCERT(id, r_j)
35: **end procedure**
36: **procedure** UPDATEINITCERT(id, r_j)
37: **if** $init\_cert_i^{id}[r_j] = \perp$ **then**
38: $x \leftarrow \{c \subseteq freezes_i^{id}[r_j] : |c| = f + 1 \wedge \forall e \in c : e.proposal = \perp \vee proposals_i^{id}[e.proposal.r] = e.proposal.v\}$
39: **if** $|x| > 0$ **then**
40: $init\_cert_i^{id}[r_j] \leftarrow$ **any element of** x
41: **end if**
42: **end if**
43: **end procedure**

Definição 5.1.1. *Um $init\_cert$ é considerado válido por um processo correto se as duas condições abaixo forem satisfeitas:*

- *Seja r' o maior r_p das mensagens FREEZE de $init\_cert$, então: qualquer duas mensagens FREEZE, m e m', pertencentes a $init\_cert$ com $rp = r'$ possui $v' = \perp$ ou $v' = v$.*
- *Todas as mensagens FREEZE do certificado possuem f+1 hashes válidos*

Definição 5.1.2. *Dado um certificado $init\_cert$, e seja r' o maior r_p das mensagens FREEZE em $inic\_cert$. O valor atestado por $init\_cert$ é:*

- *Um valor $v \neq \perp$ de uma mensagem FREEZE com $rp = r'$*
- *Se todas as mensagens FREEZE com $rp = r'$ possuem $v = \perp$, qualquer valor é atestado.*

Quando um novo líder propuser um valor, ele deve corroborar o seu valor inicial com um $init\_cert$ válido, que irá atestar que o valor proposto é válido. Da mesma forma, um processo que deseja iniciar uma nova rodada sem ter enviado uma mensagem PREPARE deve enviar mensagem corroborada por um $init\_cert$ montado na rodada anterior.

O módulo de detecção de faltas, ao detectar um processo faltoso irá descartar toda mensagem enviada por este processo a partir da mensagem que fez com que o processo . Uma possível otimização seria enviar a mensagem FREEZE imediatamente ao invés de esperar o tempo do temporizador acabar. Este módulo é o primeiro a tratar mensagens provenientes de outros processos e decide quais mensagens devem ser passadas ao restante do algoritmo.

As mensagens enviadas pelo componente confiável chegam na mesma ordem em que foram enviadas então o módulo se assegura que:

1. Mensagens possuem número de rodada sempre crescente.
2. Se o primário envia proposta do valor v na rodada r, então primário deve enviar PREAPARE com valor v.
3. Um processo qualquer que envia PREPARE com valor v na rodada r, só pode enviar nesta rodada FREEZE com valor v, rodada r e $init\_cert = \perp$.
4. Um processo que não enviou PREPARE na rodada r e deseja enviar FREEZE na mesma rodada deve fazê-lo com valor w, rodada $r_p < r_i$ e $init\_cert$ montado com mensagens da rodada $r_i - 1$.
5. As mensagens dentro de uma rodada devem ser únicas e obeceder a ordem: PROPOSE, PREPARE, FREEZE. Sendo que a mensagem PROPOSE só pode ser enviada pelo líder da rodada. Um processo pode decidir a qualquer momento, desde que possua um certificado válido

6. Após a mensagem FREEZE ser enviada na rodada r, as mensagens de PROPOSE ou o PREPARE devem pertencer à rodada $r' = r + 1$

7. Líder da rodada após enviar mensagem PROPOSE, deve enviar mensagem PREPARE antes de qualquer outra.

8. Mensagens devem possuir certificados, e assinaturas válidas quando usadas.

Qualquer violação destas regras por parte de um processo faz com que todas as mensagens subseqüentes sejam descartadas pelo módulo de detecção de falhas e não sejam recebidas pelo algoritmo. Além disso, certificados contendo mensagens deste processo em rodada maior ou igual aquela onde a violação ocorreu serão consideradas inválidas.

5.1.6 Corretude do algoritmo

Definição 5.1.3. *Um valor v é dito chaveado, se $f + 1$ mensagens PREPARE válidas certificando uma proposta v são escritas na postbox em alguma rodada r.*

Definição 5.1.4. *Uma mensagem freeze $\langle FREEZE, ass, v, r\rangle$ enviada por um processo p é válida para um processo p' se: ass é válido e, ou $v = r = \perp$ ou $v \neq \perp$ e p' leu da postbox a proposta do líder da rodada r' propondo v em r'. Mensagens de FREEZE enviadas por um processo quando ordenadas pela rodada em que foram enviadas, possuem números de rodada dos valores atestados monotonicamente crescentes.*

Definição 5.1.5. *Um $init\_cert$ válido na rodada r é composto por $f + 1$ mensagens FREEZE válidas enviadas em r por processos distintos.*

Definição 5.1.6. *Um valor v é atestato por um $init\_cert$ se uma das condições abaixo for verdadeira:*

- *todas as mensagens FREEZE pertencentes a $init\_cert$ não atestam nenhum valor.*
- *mensagem FREEZE com maior r atesta v*

Lema 5.1.1. *Se na rodada r o valor v foi chaveado, então qualquer $init\_cert$ válido na rodada r atestará v.*

Demonstração. Se um processo p qualquer envia uma mensagem PREPARE válida com valor v' em uma rodada r', então se p enviar uma mensagem FREEZE, esta deverá validar v' em r' é válida. Como o valor v foi chaveado, $f+1$ mensagens PREPARE foram escritas na postbox por processos distintos

(definição 5.1.3). Assim sendo, estes $f + 1$ processos que enviaram PREPARE poderão enviar apenas mensagens FREEZE validando v' em r'. Como existem $2f + 1$ processos, qualquer quórum de $f + 1$ mensagens de FREEZE conterá pelo menos uma mensagem de FREEZE atestando v e $r' = r$. Como não existe mensagem de FREEZE enviada em r tal que $r'' > r$ (por definição), o valor atestado pelo $init\_cert$ formado por qualquer um destes subconjuntos será v. □

Lema 5.1.2. *Se uma proposta v foi chaveada na rodada r e o líder de uma rodada $r' > r$ propor, sua proposta será v.*

Demonstração. Para provar este lema, utilizaremos indução. Para o caso base, suponha que $r' = r + 1$. Qualquer proposta feita pelo líder da rodada r' deve ser certificada por um $init\_cert$ da rodada r. Segundo o lema 5.1.1, qualquer $init\_cert$ válido montado em r certifica v. Logo, se houver uma proposta na rodada r' ela será v.

Para o passo de indução, suponha que qualquer r'', sendo que $r \leq r'' < r'$, satisfaz o lema. Pela hipótese de indução, qualquer líder de uma rodada r'', se propõe, propõe v. Logo, qualquer mensagem FREEZE válida gerada em uma rodada r'' ou certifica a proposta v feita em alguma rodada maior ou igual a r e menor ou igual a r'', ou certifica um valor qualquer proposta em uma rodada menor que r.

Como v foi congelado na rodada r, $f + 1$ processos somente poderão enviar mensagens de FREEZE validando v em r. Como o número de rodada dos valores certificados das mensagens FREEZE são monotonicamente crescentes, se estes $f + 1$ processos enviarem mensagem FREEZE válida em uma rodada maior ou igual a r, esta atestará um valor proposto em uma rodada $r'' \geq r$, ou seja, v.

Qualquer quórum de $f + 1$ mensagens de FREEZE compartilha pelo menos uma mensagem entre si. Existem $f + 1$ processos que deverão enviar mensagem de FREEZE válida atestando a proposta de uma rodada maior ou igual a r. Logo, qualquer $init\_cert$ criado em uma rodada maior ou igual a r possuirá pelo menos uma mensagem FREEZE, oriunda de um destes processos, atestando proposta de uma rodada maior ou igual a r.

Devido ao fato de que $init\_cert$ certifica o valor da mensagem FREEZE com a proposta da maior rodada e como existe pelo menos uma mensagem FREEZE atestando proposta de uma rodada maior ou igual a r, qualquer $init\_cert$ criado na rodada $r' - 1$ atestará v. Qualquer proposta feita pelo líder da rodada r' deve ser certificada por um $init\_cert$ da rodada $r' - 1$, logo qualquer proposta válida em r' será v.

Por indução concluí-se que se uma proposta v foi chaveada na rodada r então se o líder de uma rodada $r' > r$ propor, sua proposta será v. □

Lema 5.1.3. *Se todo processo correto enviou mensagem de FREEZE na rodada r, então é possível gerar um $init\_cert$ válido em r.*

Demonstração. Suponha que o lema seja falso, ou seja, todo processo correto enviou FREEZE e não é possível gerar um $init\_cert$ válido em r. Para um $init\_cert$ ser inválido ele deve possuir mensagens FREEZE válidas atestando valores não-nulos diferentes para uma mesma rodada. Mensagens FREEZE são válidas perante um processo correto p se atesta não atesta nenhum valor ou se atesta um valor v proposto pelo líder da rodada r' indicada na mensagem. Mas, devido a postbox, todo processo receberá a mesma proposta do líder na rodada r'. Assim, temos uma contradição. □

Lema 5.1.4. *Se um processo correto p decide v, então v foi chaveada.*

Demonstração. Se um processo correto p decide, ele executa a linha 29 do algoritmo 23. Para que isto ocorra, o teste da linha 28 deve ser verdadeiro, o que significa que $f + 1$ mensagens PREPARE válidas com mesmo v referentes a uma rodada r foram recebidas por p. Logo, pela definição 5.1.3, v foi chaveada na rodada r. □

Lema 5.1.5. *Se uma proposta v foi chaveada na rodada r, então o líder da rodada r propôs v.*

Demonstração. Se uma proposta v foi chaveada na rodada r, pela definição 5.1.3, sabe-se que $f + 1$ processos distintos enviaram mensagens PREPARE válidas na rodada r. Então pelo menos um processo correto enviou PREPARE. Para que um processo correto envie um PREPARE ele deve ter recebido uma mensagem do lider da rodada r com v, logo v foi proposto. □

Teorema 5.1.6. *Se dois processos corretos p_i e p_j decidem v e v', então $v = v'$*

Demonstração. Suponha que a premissa seja falsa. Logo, dois processos corretos p_i e p_j decidem v e v' e $v \neq v'$. Pelo lema 5.1.4 sabe-se que v e v' foram chaveados. Existem dois casos a considerar: v e v' foram chaveadas na mesma rodada r ou em rodadas r e r' diferentes. No primeiro caso, na rodada r $f + 1$ mensagens PREPARE atestando v e $f + 1$ mensagens atestando v' foram escritas na postbox. Como processos enviam apenas uma mensagem PREPARE por rodada, são necessários $2f + 2 > n$. Logo, não há processos suficientes para chavear ambas as propostas. No segundo caso, sabe-se que uma das duas mensagens foi chaveada antes da outra. Suponha que v tenha sido chaveada no round $r < r'$ (caso $r > r'$ o mesmo raciocínio pode ser aplicado). Pelo lema 5.1.5, o líder da rodada r' propôs v'. Mas, v foi chaveado na rodada $r < r'$ e segundo o lema 5.1.2 o líder da rodada r' propôs ou v ou

não propos nenhum valor. Como todas as possibilidades levam a contradição, podemos concluir que se dois processos corretos p_i e p_j decidem v e v', então $v = v'$. □

Teorema 5.1.7. *Todo processo correto acabará por decidir um valor.*

Demonstração. Para garantir propriedades de liveness do sistema, assume-se que o sistema apresenta períodos de síncronia durante o qual mensagens antigas chegam e timeouts não estouram. Supondo que o sistema encontra-se neste período de sincronia, existem duas possibilidades: líder faltoso e correto. Para garantir liveness, se o líder for faltoso ele deve ser trocado o que acontece através do mecanismo de troca de visão. Um líder faltoso, devido as características da postbox pode apenas não enviar sua proposta. Neste caso, os time-outs dos processos corretos estouram e mensagens de FREEZE são enviadas. Um novo líder é escolhido e inicia sua rodada. O lema 5.1.3 garante que o novo líder consiguirá propor um valor válido devido a existência de um $init\_cert$ válido na rodada anterior.

No caso do líder ser correto, ele escreverá uma mensagem PROPOSAL. Como o sistema se encontra em um período de sincronia todos os processos corretos (no mínimo $f + 1$) recebem esta mensagem do líder, enviam PREPARE e ao receber as $f + 1$ mensagens necessárias decidem.

O sistema acabará escolhendo um líder correto após, no máximo, $f + 1$ trocas de visão. Isto ocorre devido ao sistema de eleição de líder utilizado, idêntico ao utilizado em [36]. □

Teorema 5.1.8. *Se um processo correto decidir um valor v então v é um valor certificado.*

Demonstração. Para decidir um valor v, um processo p deve receber $f + 1$ mensagens PREPARE válidas com mesmo v referentes a mesma rodada r. Deste modo o teste da linha 28 é verdadeiro e a decisão da linha 29 pode ser tomada. Para uma mensagem PREPARE ser válida, um processo correto se certifica que o lídar da rodada enviou o valor atestado por este PREPARE, logo é possível concluir que houve o envio por parte do líder de uma mensagem PROPOSAL válida. Para uma mensagem PROPOSAL ser válida, v deve ser certificada. Logo pode-se concluir que se um processo correto decide v então v é certificado. □

Capítulo 6

Experimentos e Detalhes de Implementação

Neste capítulo serão apresentados alguns resultados práticos que foram obtidos através da implementação de um protótipo. Além disto, detalhes relacionados a implementação da *postbox* e da *postbox* distribuída são abordados.

6.1 Detalhes de Implementação de Postbox

Esta seção levanta alguns pontos relacionadas a implementação das duas *postbox* apresentadas. A *postbox* distribuída não chegou a ser implementada, entretanto é mostrado como implementá-la utilizando tecnologia de virtualização e TrInc [62].

6.1.1 Postbox

A *postbox* é um componente confiável que visa explorar a arquitetura da máquina virtual para prover fortes garantias de ordem de entrega de mensagens escritas e lidas da mesma. Para isso, assume-se que todos as processos participantes do consenso encontram-se na mesma máquina física, rodando isoladamente em uma máquina virtual própria. Desta forma, pode-se utilizar mecanismos como memória compartilhada com acesso de escrita controlado para implementá-la.

A figura 6.1 ilustra a arquitetura baseada em memória compartilhada para implementação da postbox utilizando uma única máquina física. Neste cenário, é responsabilidade do gerenciador de máquinas virtuais fazer com que escritas na memória compartilhada sejam atômicas e é permitido apenas anexar novas mensagens ao fim da memória compartilhada. A primeira restrição evita que mensagens enviadas por máquinas virtuais distintas se entrelacem durante a escrita, danificando a mensagem. A segunda restrição não permite que processos maliciosos de uma dada máquina virtual adulterem mensagens previamente escritas por processos corretos.

Apesar de conceitualmente simples, a implementação de tal arquitetura é um pouco complexa pois exige a implementação de um *hardware* virtual dentro do gerenciador de máquina virtual e um *driver* de acesso a este hardware para ser utilizado pelos sistemas operacionais das máquinas virtuais. Por

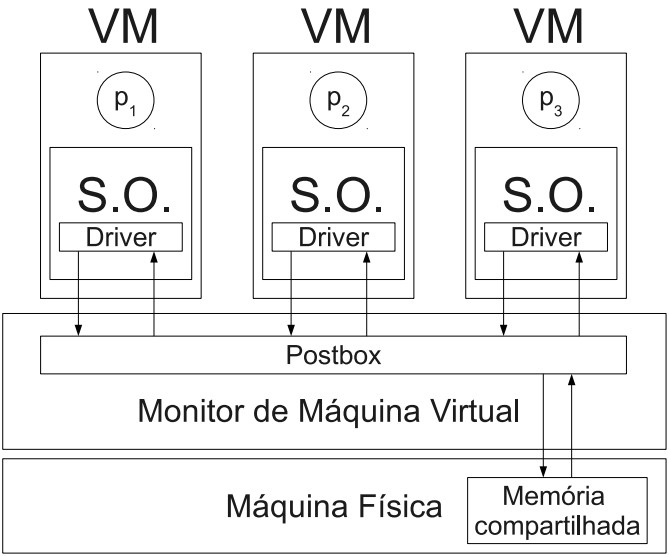


Figura 6.1: Implementação *postbox* através de memória compartilhada.

este motivo, a implementação da *postbox* adotada no protótipo é um pouco distinta.

Algumas soluções de virtualização como VirtualBox (aqui utilizado), permitem que o sistema operacional hospedeiro compartilhe arquivos e pastas com as máquinas virtuais. Este mecanismo foi adotado para a implementação da *postbox*. Ao invés de um *hardware* emulado no gerenciador de máquina virtual, associa-se a cada máquina virtual uma arquivo exclusivo. Este arquivo não pode ser lido, tampouco modificado por outras máquinas virtuais. Isto é feito através de mecanismos de controle de acesso a arquivos disponíveis no sistema operacional hospedeiro e pela configuração do gerenciador de máquinas virtuais. Além disto, um arquivo somente leitura é disponibilizado a todas as máquinas virtuais.

Os arquivos privados das máquinas virtuais são utilizados pela primitiva `Append` para o envio de uma mensagem através da *postbox*. Um processo *daemon* no hospedeiro é responsável por detectar mudanças nos arquivos privados das máquinas virtuais, ordenar as mensagens escritas e escrevê-las no arquivo público. Este arquivo público é então lido pelos processos através da primitiva `ReadFromPostbox`. Esta abordagem possui a vantagem que *drivers* para o acesso a *postbox* estão disponíveis, pois utiliza-se interface de arquivos para utilizar a mesma. Esta abordagem é ilustrada pela figura 6.2.

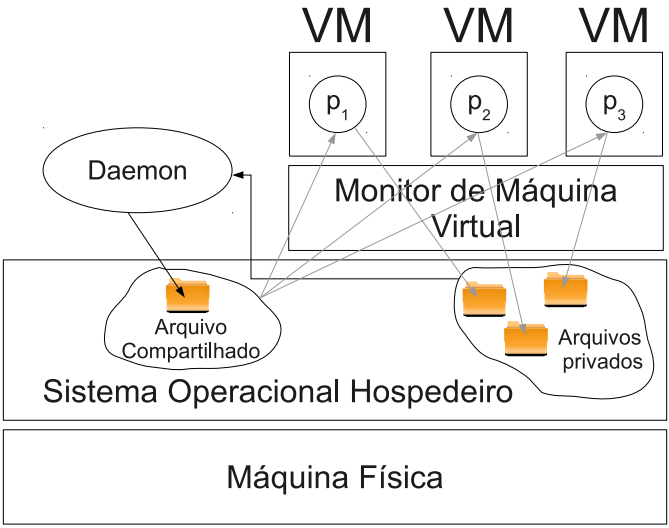


Figura 6.2: Implementação *postbox* através de arquivos compartilhados.

A *postbox* não é adequada quando ocorre a distribuição das máquinas virtuais entre diversas máquinas físicas. Esta abordagem é possível de ser implementada, entretanto o custo é alto. Isto ocorre pois todas as mensagens da postbox devem ser lidas na mesma ordem em todas as máquinas físicas, ou seja, as máquinas físicas necessitam acordar a ordem em que as mensagens serão disponibilizados às máquinas virtuais. Logo, um protocolo de consenso se faz necessário. Mesmo supondo que as *postbox* possuem um canal de comunicação privado entre si, impedindo ataques através da rede, um protocolo de acordo tolerante a faltas de *crash* se faz necessário.

6.1.2 Postbox Distribuída

A primeira proposta de implementação da *postbox* distribuída é através da utilização da tecnologia de virtualização, a exemplo do que foi feito para implementar a *postbox*

TrInc [62] utiliza um *Trusted Platform Module* (TPM) para gerar números monotonicamente crescentes. Cada componente do TrInc, chamado de *trinklet*, está disponível em todas as máquinas. O TPM é um dispositivo de segurança encontrado em muitos PCs modernos, possuindo proteções contra adulterações, que provê serviços criptográficos e contadores monotonicamente crescentes.

Algorithm 24 Implementação da *postbox* distribuída utilizando TrInc

Algorithm for p_i:

1: $counterId \leftarrow$ CREATECOUNTER
2: $counter \leftarrow 0$
3: **procedure** POSTBOXWRITE($value$)
4: $h \leftarrow$ HASH($value$)
5: $a \leftarrow Attest(counterId, counter, h)$ ▷ $a = \langle COUNTER, I, i, c, c', h\rangle$
6: $cert \leftarrow GetCertificate$
7: $counter \leftarrow counter + 1$
8: $MultiSend$ ⟨BROADCAST; $value; a; cert\rangle$ *to servers*
9: **end procedure**
10: **procedure** POSTBOXREAD(id)
11: **wait until** $nextMessage_i^{id}[pointer_i^{id}] \neq \perp$
12: $pointer_i^{id} \leftarrow pointer_i^{id} + 1$
13: **return** $nextMessage_i^{id}[pointer_i^{id} - 1]$
14: **end procedure**
15: **on receive** ⟨BROADCAST; $value; a; cert$⟩ **from** s_p
16: $h \leftarrow$ HASH($value$)
17: $valid \leftarrow messages_i^{s_p}[a.c] = \perp \wedge h = a.h \wedge$ VERIFYSIGNATURE$(a, cert.K_{pub})$
18: **if** $valid$ **then**
19: $MultiSend$ ⟨BROADCAST; $value; a; cert$⟩ *to servers*
20: $messages_i^{s_p}[a.c] \leftarrow value$
21: **end if**

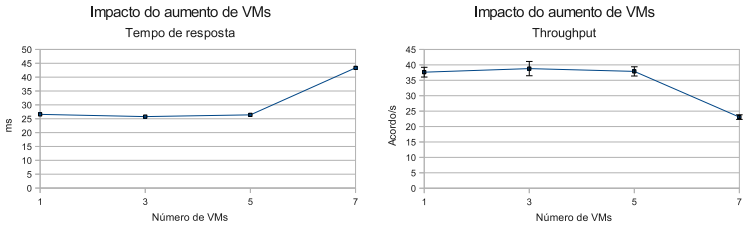
O algoritmo 24 funciona criando para cada mensagem um número autenticado pelo seu *trinklet*. Apenas mensagens com números de sequencia gerados pelo TrInc são passíveis de serem entregues. Além disto, uma mensagem m com número de sequência s só pode ser entregue se todas as mensagens com número de sequência menor que s forem entregues. Isto, aliado a uma rede ponto-a-ponto confiável, garante a implementação das propriedades exigidas pela Postbox Distribuída.

Utilizaremos as seguintes operações do TrInc:

- $CreateCounter()$ - cria um trinklet
- $Attest(i, c', h)$ - cria um atestado $\langle$COUNTER, $I, i, c, c', h\rangle_K$ onde K é a chave privada do trinklet, c' é um algum número maior que c (ultimo número atestado pelo trinklet), i é o identificador do trinklet retornado na sua criação e I é o hash da chave publica do trinklet.
- $GetCertificate()$ - retorna uma tupla $\langle I, K_{pub}, A\rangle$ contendo a chave pública deste trinklet e o hash da mesma juntamente com um atestado verificável A da validade do trinklet.
- $CheckAttestation(a, i)$ - verifica se um atestado foi gerado pelo trinklet válido identificado por i.

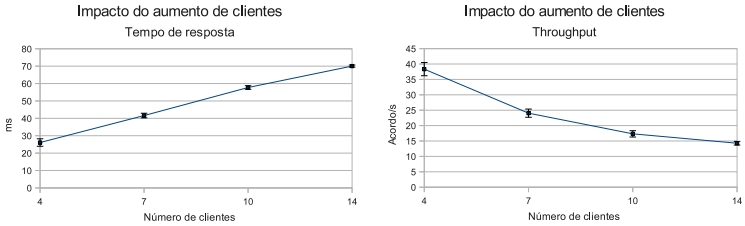
6.2 Protótipo

Para analisarmos o desempenho do sistema proposto, um protótipo foi implementado em Java 1.6. O ambiente de testes é composto por uma máquina hospedeira equipada com um CPU Quad Core 2 com 8 GB of RAM rodando Debian GNU/Linux 5.0 e VirtualBox 2.2.4. Cada máquina virtual possui acesso a 1 GB de memória RAM e 1 processador. Dentro das VMs, foi instalado um servidor rodando Ubuntu GNU/Linux 9.10. Os clientes rodam Ubuntu GNU/Linux 9.10 Desktop, sobre uma CPU Core 2 Duo com 4 GB de memória RAM. As máquinas são interconectadas através de uma rede ethernet dedicada, isto é, não há trafego de rede além do gerado pelo experimento.



(a) Tempo de resposta (b) Vazão(Throughput)

Figura 6.3: Resultado de testes com quatro clientes, variando servidores



(a) Tempo de resposta (b) Vazão(Throughput)

Figura 6.4: Resultado de testes com três servidores, variando clientes

Microbenchmarks foram realizados onde a proposta de cada cliente possui tamanho zero. Nos testes que seguem, foram realizadas entre 50 e 100 amostras, após um período de *warm-up*. Nos gráficos são mostrados o ponto médio e a marca de um desvio padrão.

No primeiro teste, o número de clientes foi constante enquanto o número

de máquinas virtuais era aumentado. O gráfico da figura 6.3(b) mostra a vazão do sistema. É possível perceber que a vazão permanece praticamente constante até atingir sete VMs; a partir deste ponto a vazão começa a diminuir. Concluímos que isto ocorre devido a limitação do poder computacional da máquina hospedeira, já que sete dos oito núcleos estão processando requisições, enquanto o núcleo restante está gerenciando a Postbox e as demais VMs. O tempo de resposta mostrado na figura 6.3(a) apresenta comportamento semelhante, começando a aumentar quando o limiar de sete máquinas virtuais é atingido.

Em outro teste foi feito o oposto: foi deixado o número de servidores constante enquanto era aumentado o número de clientes. Como pode ser visto na figura 6.4(b), a vazão neste caso diminui com o aumento do número de clientes. Isto acontece devido ao maior número de mensagens e assinaturas que devem ser processadas e verificadas. É possível notar que o aumento observado é linear, fato que pode ser explicado pelo aumento também linear do número de assinaturas. O comportamento do tempo de resposta é análogo à vazão e é mostrado na figura 6.4(a). Com o aumento de clientes o tempo de resposta começa a crescer.

Capítulo 7

Conclusão

Esta dissertação introduziu o Serviço Genérico de Consenso (SGC). O SGC é uma extensão do trabalho de serviço de consenso para suportar faltas bizantinas. O objetivo de tal serviço é prover uma plataforma única sobre a qual protocolos para resolução de problemas de acordo quaisquer possam ser desenvolvidos. Tal abordagem favorece o reaproveitamento de uma solução testada, eliminando a necessidade do desenvolvimento de protocolos para diferentes protocolos de acordo caso a caso. As generalidades são compartilhadas, enquanto as especifícidades de cada problema de acordo são inseridas através de uma interface bem definida. São desenvolvidas na dissertação a base algorítmica do Serviço Genérico de Consenso e diversos exemplos de especialização para resolver diferentes problemas de acordo.

O trabalho adota um modelo de faltas híbrido com o intuito de aumentar a resiliência dos servidores do SGC. Neste ponto, a arquitetura do SGC se prova útil. A total separação de responsabilidades imposta pelo SGC permite que os servidores adotem protocolos de acordos específicos a sua realidade, sem que com isso imponham qualquer restrição aos seus clientes. Assume-se que os servidores rodam em máquinas virtuais, sendo controladas por um monitor de máquinas virtual que provê isolamento entre as mesmas. A adoção deste modelo híbrido permite que um algoritmo de consenso tolerante a f faltas utilizando apenas $2f + 1$ VMs.

A adoção de máquinas virtuais por si só não é suficiente para o aumento da resiliência de algoritmo de consenso. Para tal, dois componentes confiáveis com características distintas foram especificados. O primeiro componentes confiável, chamado de *postbox*, foi projetado para ser implementado pelo gerenciador de máquinas virtuais sob uma única máquina física. Este componente apresenta restrições fortes em relação ao seu comportamento, com o objetivo de explorar ao máximo as possibilidades da máquina virtual, no caso, memórias compartilhadas. Entretanto, isto a torna vulnerável ao *crash* da máquina hospedeira. O segundo componente, chamado de *postbox* distribuída, relaxa algumas restrições da *postbox* facilitando a distribuição deste componente entre diversas máquinas físicas. Para isto, assume-se que estes

componentes confiáveis gozam de uma rede privada de comunicação que não permite ataques externos.

Visando tirar proveito das características dos componentes confiáveis, dois algoritmos de consenso foram desenvolvidos. O primeiro algoritmo de consenso é baseado na *postbox*, sendo um algoritmo sem líder capaz de resolver o consenso em apenas um passo de comunicação através da *postbox*. A sua principal desvantagem se encontra na vulnerabilidade à falha de *crash* do servidor. O segundo algoritmo de consenso, baseado na *postbox* distribuída, é um algoritmo baseado em líder, decidindo em dois passos de comunicação em uma execução sem faltas. Este algoritmo, apesar de tolerar a falta de crash do hospedeiro, é muito mais complexo que o primeiro.

Finalmente, mostramos alguns resultados obtidos durante a execução de um pequeno protótipo em um ambiente controlado.

Dentre trabalhos futuros que poderão ser desenvolvidos encontram-se:

- Estudo de outros modelos não-virtualizados para a implementação do SGC, mantendo a mesma resiliência obtida neste trabalho
- Desenvolvimento de outros algoritmos para resolução de problemas de acordo baseados no SGC. Dentre problemas interessantes não abordados nesta dissertação é possível citar o problema da Sincronia Virtual.
- Proposição de novos componentes confiáveis, que levem a algoritmos mais simples ou benefícios de implementação.
- Implementação e testes para avaliar o comportamento do SGC baseado na *postbox* distribuída.
- Possibilidade do conjunto de clientes ser dinâmico. Isto permitiria que protocolos como *group membership* fossem implementados sem que a distinção entre coordenadores e clientes seja feita.

Referências Bibliográficas

[1] SCHNEIDER, F. B.; LAMPORT, L. Paradigms for distributed programs. In: *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*. London, UK: Springer-Verlag, 1985. p. 431–480. ISBN 3-540-15216-4.

[2] CORREIA, M.; NEVES, N. F.; VERÍSSIMO, P. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *Comput. J.*, v. 49, n. 1, p. 82–96, 2006. ISSN 0010-4620. 1183871.

[3] ZIELIŃSKI, P. *Paxos at war*. [S.l.], 2004. Disponível em: <http://www.cl.cam.ac.uk/techreports/ UCAM-CL-TR-593.pdf>.

[4] CHUN, B. et al. Attested append-only memory: making adversaries stick to their word. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. Stevenson, Washington, USA: ACM, 2007. p. 189–204. ISBN 978-1-59593-591-5.

[5] CORREIA, M. et al. Low complexity byzantine-resilient consensus. *Distrib. Comput.*, v. 17, n. 3, p. 237–249, 2005. ISSN 0178-2770. 1151559.

[6] CHUN, B.; MANIATIS, P.; SHENKER, S. Diverse replication for single-machine byzantine-fault tolerance. In: *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Boston, Massachusetts: USENIX Association, 2008. p. 287–292.

[7] REISER, H. P.; KAPITZA, R. Fault and intrusion tolerance on the basis of virtual machines. *Tagungsband des 1. Fachgesprch Virtualisierung*, 2008.

[8] REISER, H. P.; KAPITZA, R. VM-FIT: supporting intrusion tolerance with virtualisation technology. In: *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems*. [S.l.: s.n.], 2007.

[9] HILTUNEN, M. A.; SCHLICHTING, R. D.; UGARTE, C. A. Building survivable services using redundancy and adaptation. *IEEE Trans. Com-*

put., IEEE Computer Society, Washington, DC, USA, v. 52, n. 2, p. 181–194, 2003. ISSN 0018-9340.

[10] BESSANI, A. N. et al. *Intrusion-tolerant protection for critical infrastructures*. [S.l.], 2007.

[11] GUERRAOUI, R.; SCHIPER, A. The generic consensus service. *IEEE Trans. Softw. Eng.*, v. 27, n. 1, p. 29–41, 2001. ISSN 0098-5589. 359565.

[12] SCRIMGER, R.; LaSalle, P.; PARIHAR, M. *TCP/IP-A BIBLIA*. [S.l.]: Campus/Elsevier, 2002.

[13] TSUDIK, G. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, v. 22, n. 5, p. 29–38, 1992.

[14] STINSON, D. R. *Cryptography: theory and practice*. [S.l.]: CRC press, 2006.

[15] DIFFIE, W.; HELLMAN, M. New directions in cryptography. *IEEE Transactions on information Theory*, v. 22, n. 6, p. 644–654, 1976.

[16] DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, v. 35, n. 2, p. 288–323, 1988.

[17] DOLEV, D.; DWORK, C.; STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, v. 34, n. 1, p. 77–97, 1987.

[18] FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM*, v. 32, n. 2, p. 374–382, 1985.

[19] HADZILACOS, V.; TOUEG, S. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Ithaca, NY, USA, 1994.

[20] CORREIA, M. et al. Efficient Byzantine-Resilient reliable multicast on a hybrid failure model. In: *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*. [S.l.]: IEEE Computer Society, 2002. p. 2. ISBN 0-7695-1659-9.

[21] HURFIN, M. et al. A general framework to solve agreement problems. In: *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*. Lausanne, Switzerland: [s.n.]. p. 56–65.

[22] CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM*, v. 43, n. 2, p. 225–267, 1996. ISSN 0004-5411. 226647.

[23] CHARRON-BOST, B.; SCHIPER, A. Uniform consensus is harder than consensus. *Journal of Algorithms*, v. 51, n. 1, p. 15–37, 2004.

[24] DOUDOU, A. et al. Muteness failure detectors: Specification and implementation. In: *Proceedings of the Third European Dependable Computing Conference on Dependable Computing*. [S.l.]: Springer-Verlag, 1999. p. 71–87. ISBN 3-540-66483-1.

[25] MALKHI, D.; REITER, M. Unreliable intrusion detection in distributed computations. In: *Proceedings of the 10th IEEE Computer Security Foundations Workshop*. [S.l.: s.n.], 1997. p. 116–124.

[26] LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, v. 4, n. 3, p. 382–401, 1982. ISSN 0164-0925. 357176.

[27] CACHIN, C. et al. Secure and efficient asynchronous broadcast protocols. In: *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*. London, UK: Springer-Verlag, 2001. p. 524–541. ISBN 3-540-42456-3.

[28] REITER, M. K. A secure group membership protocol. *IEEE Transactions on Software Engineering*, v. 22, n. 1, p. 31–42, 1996.

[29] GUERRAOUI, R. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distrib. Comput.*, Springer-Verlag, London, UK, v. 15, n. 1, p. 17–25, 2002. ISSN 0178-2770.

[30] DOUDOU, A.; GARBINATO, B.; GUERRAOUI, R. Encapsulating failure detection: From crash to byzantine failures. *Lecture notes in computer science*, Springer-Verlag, p. 24–50, 2002.

[31] REYNAL, M. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, v. 36, n. 1, p. 53–70, 2005. ISSN 0163-5700. 1052806.

[32] BERNSTEIN, P. A.; NEWCOMER, E. *Principles of transaction processing*. [S.l.]: Morgan Kaufmann Pub, 2009.

[33] GRAY, J. Notes on data base operating systems. In: *Operating Systems, An Advanced Course*. [S.l.]: Springer-Verlag, 1978. p. 393–481. ISBN 3-540-08755-9.

[34] GRAY, J.; LAMPORT, L. Consensus on transaction commit. *ACM Trans. Database Syst.*, v. 31, n. 1, p. 133–160, 2006.

[35] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, v. 22, n. 4, p. 299–319, 1990.

[36] CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, v. 20, n. 4, p. 398–461, 2002. ISSN 0734-2071.

[37] NANDA, S.; CHIUEH, T. A survey on virtualization technologies. *Stony Brook University, Tech. Rep*, v. 179, 2005.

[38] ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: current technology and future trends. *Computer*, v. 38, n. 5, p. 39–47, 2005. ISSN 0018-9162.

[39] CHAVES, J. Enabling high productivity computing through virtualization. In: *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*. [S.l.: s.n.], 2008. p. 403–408.

[40] BARHAM, P. et al. Xen and the art of virtualization. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. Bolton Landing, NY, USA: ACM, 2003. p. 164–177. ISBN 1-58113-757-5.

[41] MICROSYSTEMS, S. *Virtualbox*. 2010. Http://www.virtualbox.org.

[42] SUGERMAN, J.; VENKITACHALAM, G.; LIM, B. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. [S.l.]: USENIX Association, 2001. p. 1–14. ISBN 1-880446-09-X.

[43] KIHLSTROM, K. P.; MOSER, L. E.; Melliar-Smith, P. M. Solving consensus in a byzantine environment using an unreliable fault detector. In: *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*. [S.l.: s.n.], 1997. p. 61–75.

[44] SCHIPER, A. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, v. 10, n. 3, p. 149–157, 1997.

[45] GUERRAOUI, R. et al. Consensus in asynchronous distributed systems: A concise guided tour. *Lecture Notes in Computer Science*, p. 33–47, 2000.

[46] DOUDOU, A.; SCHIPER, A. Muteness detectors for consensus with byzantine processes. In: *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*. Puerto Vallarta, Mexico: ACM, 1998. p. 315. ISBN 0-89791-977-7.

[47] LAMPORT, L. Paxos made simple. *ACM SIGACT News*, v. 32, n. 4, p. 18–25, 2001.

[48] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, v. 16, n. 2, p. 133–169, 1998.

[49] LAMPORT, L. Fast paxos. *Distributed Computing*, v. 19, n. 2, p. 79–103, out. 2006.

[50] LAMPORT, L.; MASSA, M. Cheap paxos. In: *Proceedings of the 2004 International Conference on Dependable Systems and Networks*. [S.l.]: IEEE Computer Society, 2004. p. 307. ISBN 0-7695-2052-9.

[51] GAFNI, E.; LAMPORT, L. Disk paxos. *Distrib. Comput.*, v. 16, n. 1, p. 1–20, 2003.

[52] CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: an engineering perspective. In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. Portland, Oregon, USA: ACM, 2007. p. 398–407. ISBN 978-1-59593-616-5.

[53] PRISCO, R. D.; LAMPSON, B. W.; LYNCH, N. A. Revisiting the paxos algorithm. In: *Proceedings of the 11th International Workshop on Distributed Algorithms*. [S.l.]: Springer-Verlag, 1997. p. 111–125. ISBN 3-540-63575-0.

[54] MALKHI, D.; OPREA, F.; ZHOU, L. Ω meets paxos: Leader election and stability without eventual timely links. In: *Distributed Computing*. [S.l.: s.n.], 2005. p. 199–213.

[55] MALKHI, D.; REITER, M. Byzantine quorum systems. *Distrib. Comput.*, v. 11, n. 4, p. 203–213, 1998. ISSN 0178-2770. 1035782.

[56] KIHLSTROM, K. P.; MOSER, L. E.; Melliar-Smith, P. M. The SecureRing protocols for securing group communication. In: *Hawaii International Conference on System Sciences*. Los Alamitos, CA, USA: IEEE Computer Society, 1998. v. 3, p. 317.

[57] KOTLA, R. et al. Zyzzyva: speculative byzantine fault tolerance. *Commun. ACM*, v. 51, n. 11, p. 86–95, 2008.

[58] VERONESE, G. S. et al. Spin one's wheels? byzantine fault tolerance with a spinning primary. In: *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*. [S.l.]: IEEE Computer Society, 2009. p. 135–144. ISBN 978-0-7695-3826-6.

[59] HERLIHY, M. P.; WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, v. 12, n. 3, p. 463–492, 1990.

[60] YIN, J. et al. Separating agreement from execution for byzantine fault tolerant services. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. Bolton Landing, NY, USA: ACM, 2003. p. 253–267. ISBN 1-58113-757-5.

[61] CASTRO, M. O. T. D. *Practical byzantine fault tolerance*. 1 p. Tese (Doutorado), 2000. 935468 Supervisor - Barbara H. Liskov.

[62] LEVIN, D. et al. Trinc: small trusted hardware for large distributed systems. In: *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2009. p. 1–14.

[63] TOMLINSON, A. Introduction to the TPM. In: *Smart Cards, Tokens, Security and Applications*. [S.l.: s.n.], 2008. p. 155–172.

[64] KINNEY, S. *Trusted platform module basics: using TPM in embedded systems*. [S.l.]: Newnes, 2006.

[65] Abd-El-Malek, M. et al. Fault-scalable byzantine fault-tolerant services. In: *Proceedings of the twentieth ACM symposium on Operating systems principles*. Brighton, United Kingdom: ACM, 2005. p. 59–74. ISBN 1-59593-079-5.

[66] LI, J. et al. Secure untrusted data repository (SUNDR). In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design \& Implementation - Volume 6*. San Francisco, CA: USENIX Association, 2004. p. 9–9.

[67] OBELHEIRO, R. R. et al. *How practical are intrusion-tolerant distributed systems?* [S.l.], 2006.

[68] CLEMENT, A. et al. BFT: the time is now. In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. Yorktown Heights, New York: ACM, 2008. p. 1–4. ISBN 978-1-60558-296-2.

[69] AMIR, Y. et al. Byzantine replication under attack. In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. [S.l.: s.n.], 2008. p. 197–206.

[70] CLEMENT, A. et al. Making byzantine fault tolerant systems tolerate byzantine faults. In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. Boston, Massachusetts: USENIX Association, 2009. p. 153–168.

[71] REISER, H. P.; DISTLER, T.; KAPITZA, R. Functional decomposition and interactions in hybrid intrusion-tolerant systems. In: *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction*. Lisbon, Portugal: ACM, 2009. p. 7–12. ISBN 978-1-60558-489-8.

[72] CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. In: *Proceedings of the third symposium on Operating systems design and implementation*. New Orleans, Louisiana, United States: USENIX Association, 1999. p. 173–186. ISBN 1-880446-39-1.